Inducing game rules from varying quality game play

Alastair Flynn

April 8, 2020

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 these game descriptions 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 proved to be a promising approach to this problem though it has been shown 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 or in connect 4 not being able to put a counter into a full column. 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, in this paper we analyse the effect of using optimal 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 all combinations of optimal and random data. We also vary the volume of training data.

AC: Our results show [can fill in later]

AC: The implications of this work are [can fill in later]

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.[14]. Traditional game playing AI has focused on a single game. Famouse AI such as IBMs deep blue is able to beat grand masters at chess but is completetly unable to play checkers. These traditional AI also only do part of the work. A lot of the analysis of the game is done outside of the system. A more interesting challenge is building AI that can play games without any prior knowlegde. 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 relativly simple Rock Paper Scissors to Chess[13]. The progress in the field is consolidated annually at the GGP where participants compete to find the best GGP AI.

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. The games are specified in the Game Description Language (GDL), a logic programming language built for describing games as state machines[23]. A logic programming language being any that is mainly based on formal logic, such as Prolog.

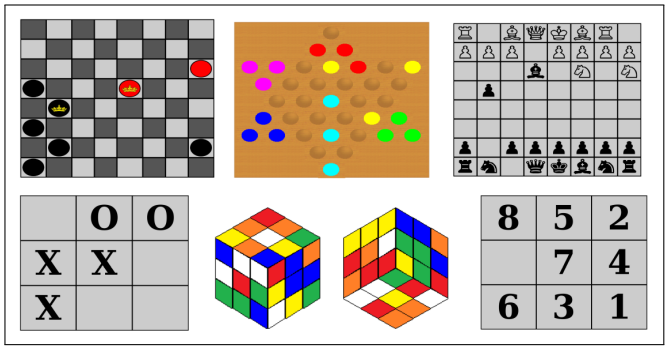


Figure 1.1: A selection of games from the GGP competition. From the top right: *checkers*, *chinese checkers*, *chess*, *tic tac toe*, *rubix cube* and *eight puzzle*

These GDL game descriptions form the basis for the Inductive General Game

```
(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

Playing (IGGP) problem. 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[6]. The games used in IGGP are those from the GGP competition problem set, specified in GDL, meaning they are widely varied in complexity.

AF: GIVE EXAMPLE OF IGGP PROBLEM

An effective way of solving the IGGP problem is a form of machine learning: Inductive Logic Programming (ILP). 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 [6], the authors showed through empirical evaluations that ILP systems achieve the best score in this task compared to other machine learning techniques. The ILP system derive a hypothesis, a logic program that when combined with the background knowledge, entails all of the positive and none of the negative examples[25]. In the IGGP paper[6] 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.

However, the existing work has limitations. All existing work has assumed that the gameplay being observed is randomly generated. 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 terminal 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 weather 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 sets of gameplay observations - optimal gameplay verses random gameplay as well as combinations of the two. We also vary the number of

gameplay traces from which the ILP systems learn.

It is not obvious whether random or optimal gameplay would be best. 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 optimal 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 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 is an easier question to 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. 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 [25]. 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 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 $5.0 \cdot 10^{20}$[29] and sudoku which exceeds $6.6 \cdot 10^{21}$[11]. Instead we will test the learned programs on both optimal and random data of the same quality used in training. AF: and maybe a combination as well

We would also 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 [24]. However, Gonzales and Abu-Mostafa[15] 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[1] 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[15]. 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 [1]. 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, AF: (2) world-leading, and (3) optimal levels. (Section 6)
- We transform the GGP traces to IGGP problems
- We train the ILP systems Metagol, Aleph and ILASP on different combinations of optimal and random data as well as testing them on each individually (Section 6)
- 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

[1] 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 [6]. 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. The games also vary in number of players, dimensions and complexy. For example games such as rock paper scissons have 0 dimensions and only 10 rules in the given GGP ruleset, 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 send a listing of the rules descirbed in the Game Description Language (GDL). GDL is a language based on first order logic described in section 2.1.1. The list of games along with their descriptions are available online [1]. Matches in the GGP framework take place through an online framework called the Gamemaster. The agents connect to an online Game Manager (part of the Gamemaster) service which arbitrates indavidual matches. The connecting agents send several pieces of information to the Game Manger incliding a game that is already know by the manager and the required number of players. The match can then be started by pressing a start button, when pressed the agents will recive a *Start* message including the role of the player (e.g. black or white in chess) and a description on the game in GDL. The Game Manger communicates all instructions to the players through HTTP[14].

In 2005 an annual International General Game Playing Competition (IGGPC) was set up which still runs to this day[18]. 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. The game descriptions written in GDL used in the GGP competition can be used as ideal rule sets for systems in Inductive General Game Playing (IGGP) to generate. The descriptions of the games used in GGP are not neces-

[1] http://ggp.stanford.edu/iggpc/ accessed on 17/03/2020

sarily minimal so it is possible that an ILP system could generate a more concise ruleset the 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.[23] The language is based off logical programming languages such as Prolog and is in fact a logical programming language itself.

Logical 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. 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. It 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

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

2.2 Prolog

Prolog is one of the most popular and well used logical programming languages. The syntax in Prolog for rules and facts is 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 ($\Leftarrow$). 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)`.

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 [16]. 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 are all roughly this algorithm.

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 wearched 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, assignes the last variable a differnt values. This continues until a proof is found or all possibilities have been exauseted[2].

2.3 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[6]. IGGP was designed as a way of benchmarking machine learning systems.

The definition of task itself is based on the Inductive logic programming problem.

2.3.1 ILP

Inductive logic programming is a form of machine learning that uses logic programming to represent examples, background knowledge, and learned programs[9]. 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

$$B \wedge H \vDash O$$

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

AF: Example task (grandparent relation)

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 often $B \wedge H \vDash O$ in the observations O. When all the observations are implied then a correct program has been found.

2.3.2 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 2.3.1 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). An example of a typical IGGP problem is given below

The IGGP problem itself is described in section **??**.

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.

Chapter 3

Logical setting

The IGGP problem is defined in the 2019 paper Inductive General Game Playing [6]. Much like the problem of ILP 2.3.1 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 described in 2.3.1. All the predicates involved in the task are taken from the GDL descriptions of games in the Stanford GGP* library. A lot of the atoms in the descriptions are not function-free, that is, they are nested predicates. For example `true(count(9))`. We flatten all of these to single, non nested predicates, i.e. `true_count(9)`. This is done because some ILP systems do not 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[27]. 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 the use it to define the IGGP problem. An IGGP input needs to capture the idea of an observation about a game. The input is based on the general input for the Logical induction problem of section 2.3.1 since this is a sub problem of it.

The IGGP Input: An input Δ is a set of triples $\{(B_i, E_i^+, E_i^-)\}_m^{i=1}$ where

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

An IGGP input forms the IGGP problem:

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 $H \cup B_i \not\models E_i^-$.

Problem Setting

Let the accuracy of a set I of ILP systems in problem setting be defined as the mean of the percentage accuracy of each of them when tested on a given set of examples.

Given an set of ILP systems I, for what selection of game traces Δ combined from an optimal gameplay distribution and a random gameplay distribution are the systems most accurate when solving the IGGP problem. The accuracy of I when solving the IGGP problem will be tested by evaluating each $i \in I$ with data taken from a 50/50 combination of both optimal and random distributions.

Also given a set of ILP systems I, and a gameplay distribution G how does the number of training samples $|\Delta|$ taken from G affect the accuracy of the hypothesis H for each $i \in I$ when tested on G.

3.1 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 [2]. Some small modifications to the information logged by the game server are made but otherwise the code is unchanged.

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

Monte Carlo Tree Search

Given a game state the basic MCTS will return the most promising next move. The algorithm achives 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[28][3].

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 game and the children of each node being all the states that can be moved to from that state. 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

[1] http://ggp.stanford.edu/iggpc/winners.php accessed on 12/03/2020
[2] http://sanchoggp.github.io/sancho-ggp/ access 12/03/2020

unvisited child. 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 more promising moves are investigated more.

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[17]. 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. $\frac{Q(v_i)}{N(v_i)}$ 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. $c\sqrt{\frac{ln(N(v_i))}{N(v_i)}}$ 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.

Sancho makes a few modifications to MCTS[3], the main one being 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 other modifications 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 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, or 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.

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

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.

3.1.1 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 analyised using puzzle solving techniques.

Puzzle Solving

A single player game in the GGP dataset is neciserally 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 constraint that any solution found in a playout of the game will always remain valid since the game is completely deterministic. Sancho identifies these single player games and attempts to derive huristics. Where an obviouse goal state exsists 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 admisable huristic so sancho finds an optimal solution.

Sancho also does static analysis of the game rules. It determins game predicates that if true mean the goal will never be true in any proceding state as well as predicates that garentee that the goal will be true in a future state. An example of this is a game such as untwisty corridor (the game is effecitvly a maze where if you immidiatly lose if you step off a 'safe' path). If the safe path is left then the goal can never be reached so any state not on the safe path is avoided.

AF: Talk about why this makes sancho good for this task

Chapter 4

ILP systems used

We use three ILP systems to compare the effects of different learning data, Metagol, Aleph and ILASP. All three systems use different approaches the ILP problem.

4.1 Metagol

The Metagol ILP system is a meta-interpreter for Prolog, that is, it is written in the same language is evaluates. 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)
```

[**?**][7][8]

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[1]. If no BK can be found that prove the positive example then try adding a new invented predicate and attempt to prove this[2].

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 [5]. For this task a set of 9 derivationally irreducible metarules are used which remain consistent across all tasks.

4.2 ALEPH

Aleph is an Prolog variant of the ILP system Progol [26]. 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 [10] 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 [3]:

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 [26]) that entails the example selected and is within language restrictions provided.

[1] 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

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

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

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.

4.3 ILASP

ILASP is an ILP system based on Answer Set Programming (ASP). An introduction to ASP can be found here [4]. ILASP uses a subset of ASP that is defined in these papers[19][21][20] 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[12] to check which rules should be active if the program is to be consistent with the positive and the negation of the negative examples[21][20]. In this paper we use a version of ILASP based on ILASP2i[22] which was developed with the IGGP problem in mind[6]. 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 [6]. We expect it to do well in the experiments conducted.

Chapter 5

Implementations

5.1 Eight Puzzle

To write an optimal player for Eight Puzzle I decide to use an approach based on best first search.

5.1.1 Best first search

Best first search generates a graph of the game states, expanding the most promising nodes first according to a heuristic for how close to the goal state the current state is. When the goal node is reached the search gives the path from the root node to it, i.e. the list of moves made to get to the goal. Since Best first search is a well know and often used algorithm in Prolog we decided to use a standard implmentation from Bratko[2]. This decision was made to avoid needless mistakes and inefficiency that would have come with a new independent implementation of this algorithm.

Best First search is a generic algorithm that needs certain problem specific predicates to be implemented for it to function. These predicates are:

- **The successor predicate -** `s(Node,Node1,Cost)`: This predicate is true if there is an arc costing `Cost` between state `Node` and state `Node1`. In Eight Puzzle we set the cost of all arcs to be 1.
- **The Goal predicate -** `goal(Node)`: This is true if the state `Node` is a goal state.
- **The heuristic -** `h(Node,H)`: This is a relation that relates a state `Node` to a value `H` that is a heuristic for how close to the state is to the goal state.

To represent the board I decided to use a 9 element list with `b` representing the blank tile INSERT IMAGE OF GOAL BOARD

```
[1,2,3,4,5,6,7,8,b]
```

. I defined some useful helper predicates to start with. The Manhatten distance between two tiles between two tiles is the difference in X coordinates plus the difference in Y coordinates, I wrote a predicate to calculate this relation:

```
mandist(X1-Y1,X2-Y2,D) :-
    diff(X1,X2,Dx),
    diff(Y1,Y2,Dy),
    D is Dx+Dy.
```

I also wrote predicates to calculate the coordinates of a tile from its position in the list, one that relates two tiles if they are next to each other on the grid and one that gives the position in the list of the blank tile.

Successor

To define the successor predicate I decided the best way to do it was to relate two boards by swapping the blank tile with its neighbours. I first wrote a function swap:

```
swap(I,J,L1,L3) :-
   same_length(L1,L3),
   append(BeforeI,[AtI|PastI],L1),
   append(BeforeI,[AtJ|PastI],L2),
   append(BeforeJ,[AtJ|PastJ],L2),
   append(BeforeJ,[AtI|PastJ],L3),
   length(BeforeI,I),
   length(BeforeJ,J).
```

This function only uses built in predicates that all do the obvious thing. In the first two appends this rule takes L1 and replaces the element I with the element J to make list L2 then in the second two takes list L2 and replaces J with I to make list L3. The variable AtI and AtJ will be instantiated with the elements at index I and J because of the restriction on the length of the sublist before them at the end of the rule. The successor function is defined as:

```
s(B1,B2,1) :-
    member(N,B1),
    nth0(NI,B1,N),
    blank_pos(B1,BI),
    neighbor(BI,NI),
    swap(BI,NI,B1,B2).
```

Since all arc (move) costs in my version of Eight Puzzle are 1 the predicate is only true when Cost is 1. `member(N,B1)` is true when N is a member of B1. Here it restricts N to values on the board. `nth0(NI,B1,N)` sets NI to the index of the value N in the list B1, i.e. it tells us the position on the board of the value we are considering. `blank_pos(B1,BI)` sets BI to the position on the board of the blank. `neighbor(BI,NI)` is true if the positions BI and NI are next to each other on the board. If all of these are true we then instantiate B2 with the new board by swapping the blank and its neighbour.

Goal

Defining the goal was actually very simple, I added the predicate:

```
goal([1,2,3,4,5,6,7,8,b])
```

This is all that is needed to define the goal state.

Heuristic

A good heuristic for Eight Puzzle needs to capture an estimate of the distance of the current state from the goal. I initially decided to use the sum of the Manhattan distances of each tile from its position in the goal state. To calculate this I wrote a predicate `totdist(L,Goal,N)` which relates a board L, a goal Goal and the sum of the Manhattan distances N.

```
totdist(L,Goal,N) :-
    totdist1(L,Goal,N,0).

totdist1([],_Goal,0,_Pos).

totdist1([b|T],Goal,N,Pos) :-
    !,Pos1 is Pos + 1,
    totdist1(T,Goal,N,Pos1).

totdist1([V|T],Goal,N,Pos) :-
    nth0(I,Goal,V),
    coord(Pos,X1-Y1),
    coord(I,X2-Y2),
    mandist(X1-Y1,X2-Y2,D),
    Pos1 is Pos + 1,
    totdist1(T,Goal,N1,Pos1),
    N is N1 + D.
```

My implementation of this predicate iterates through the list working out the Manhattan distance for each tile. When iterating through the list the way to keep track of how far through it you are (the position on the board you are currently looking at) is use a separate variable as a counter. Here I used Pos. After testing my program with this heuristic I found that it took about 6 seconds to come up with the first solution. I decided that I could probably do better if I had another heuristic that I combined with the Manhattan distance. I decided to use the number of tiles that were out of the row and/or column they were in. I wrote a predicate `outofpos(B,Goal,N,Pos)` that relates the board and the goal to the number of tiles out column plus the number of tiles out of row.

```
outofpos([],_Goal,0,_Pos).

outofpos([b|T],Goal,N,Pos) :- !,
    Pos1 is Pos + 1,
    outofpos(T,Goal,N,Pos1).

outofpos([V|T],Goal,N,Pos) :-
    coord(Pos,X1-Y1),
    nth0(PosG,Goal,V),
    coord(PosG,X2-Y2),
    Pos1 is Pos + 1,
    outofpos(T,Goal,N1,Pos1),
    (   xandy(X1 = X2, Y1 = Y2) ->
        N is N1
```

```
    ;   xory(X1 = X2, Y1 = Y2) ->
        N is N1 + 1
    ;   N is N1 + 2
    ).

xandy(X,Y) :- X,Y.
xory(X,Y) :- X;Y.
```

This predicate iterates through the board and for each tile looks at the place it is on the goal board and checks which coordinates match. If both match then it is the right column and row so the total remains the same, otherwise it is incremented. I eventually found that the heuristic the worked the fastest was Manhattan distance total + 2 * number of tiles out of position. I wrote this predicate for the heuristic:

```
h(B,H) :-
    goal(Goal),
    totdist(B,Goal,D),
    outofpos(B,Goal,N,0),
    H is D + (2*N).
```

5.2 Noughts and crosses

Chapter 6

Experiments

6.1 Optimal Play vs Random with fixed sample size

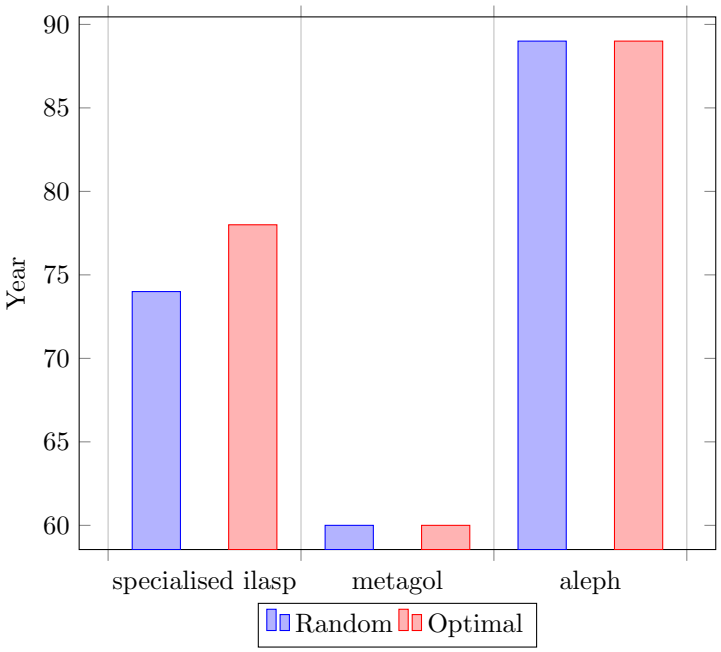
6.2 Mixed datasets (50/50 random and optimal)

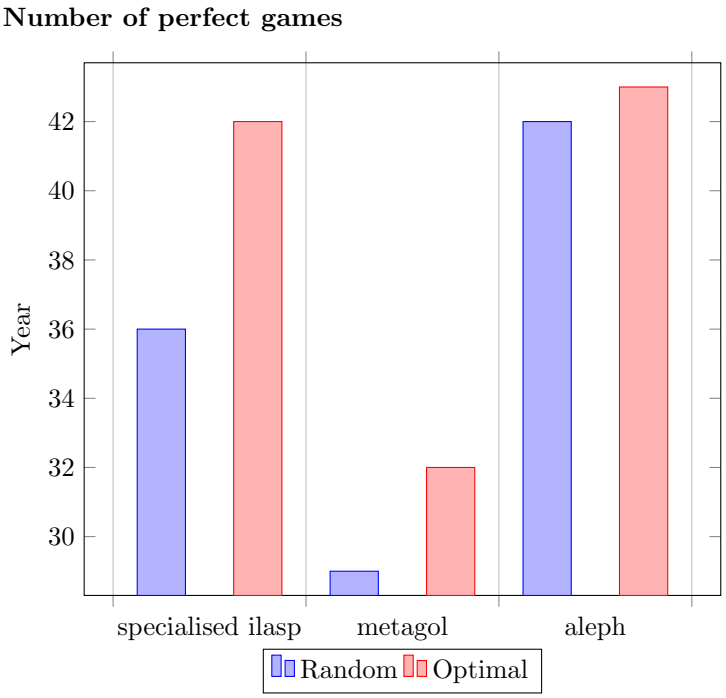
6.2.1 Results

Training Data	Testing Data	Predicate	Metagol	Aleph

Random training and testing

Balanced Accuracy





6.3 Optimal Play vs Random with varying sample size

Chapter 7

Results

Bibliography

[1] 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.

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

[3] 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.

[4] 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.

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

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

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

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

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

[10] 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.

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

[12] 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.

[13] Michael Genesereth. Gdl website.

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

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

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

[17] 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.

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

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

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

[21] 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.

[22] 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.

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

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

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

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

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

[28] 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.

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