Research tool for AI search in graphs

Student Name: Alexander Noske

Supervisor Name: Andrei Krokhin

Submitted as part of the degree of MEng Computer Science to the

Board of Examiners in the Department of Computer Science, Durham University.

Abstract –

Context/Background – The constraint satisfaction problem is key to the development of computer science as many natural combinatorial problems can be expressed as constraint satisfaction problems. This set of problems is known to be NP-complete in general, but certain restrictions on the constraints can ensure tractability. These restrictions involve whether digraphs depicting a constraint satisfaction problem admit certain polymorphisms. Thus, finding if a given pair of digraphs admits certain polymorphisms is the key to determining the tractability of certain constraint satisfaction problems.

Aims – The aim for this project is to investigate methods and techniques for an implementation of a flexible and usable software tool (PolyFind) to search for special polymorphisms between digraphs. The project also aims to evaluate and optimize the performance of this tool and compare its efficiency to other current alternatives.

Method – A python script which transforms the problem of whether a certain polymorphism exists between two digraphs into first order logic was implemented. This script was made more efficient by a variety of modelling techniques such as looking at the inherent symmetry of the problem. This first order logic is then passed to the first order model finder Paradox, which then returns whether a certain polymorphism is present or not.

Results – The produced software tool PolyFind is flexible, intuitive, and deals with a wide range of polymorphism types as well as large digraph sizes. Performance optimization is achieved by using modelling techniques. When compared to the alternatives PolyFind outperforms any of the found current alternatives by a significant margine.

Conclusion – Finding polymorphisms, while a computationally difficult task can be tackled in a multitude of ways, making the problem more approachable. As the problem of finding polymorphisms is linked to the tractability of constraint satisfaction problems, determining if a given constraint satisfaction problem is tractable (or not) is therefore also a more accessible problem.

Keywords – Polymorphisms, Search, Graphs, Constraint Satisfaction Problem, Tractability

I. INTRODUCTION

The Constraint Satisfaction Problem (CSP) is one of few problems central to the development of theoretical computer science. One of the most significant motivations for studying CSP is its applicability – it provides a common framework for many combinatorial problems in artificial intelligence and applied computer science [17]. In a constraint satisfaction problem, the aim is to find an assignment of values to a given set of variables, subject to domains and constraints on the values which can be assigned simultaneously to certain specified subsets of the variables [9].

Due to the constraint satisfaction problems applicability to a multitude of problems in industry and theoretical computer science, it is a very active research area. Specialized solvers have been built in order to solve this problem efficiently. The paper Basic Techniques for Creating an Efficient CSP Solver [13] mentions a few [14,16,12,10]. However, the constraint satisfaction problem is a challenging problem to solve quickly as it is known, in general, to be NP-complete [9]. Throughout the paper we assume that P does not equal NP, and we call a problem tractable only if it belongs to P. While generally the constraint satisfaction problem is not tractable, placing certain restrictions on the constraint satisfaction problem has been shown to ensure tractability. This restriction involves specifying a constraint language, that is a set of relations that can be used as constraints, effectively limiting the allowed constraints. However, there is still the problem of distinguishing those constraint languages which give rise to tractable problems, from those which do not. In order to explain how to do so, first the constraint satisfaction problem needs to be explained with respect to a digraph. The constraint satisfaction problem can be described as for a fixed finite digraph A, the constraint satisfaction problem over A, CSP(A) for short, can be formulated as a decision problem asking whether an input (finite) relational structure X of the same type as A admits a homomorphism to A. This description of CSP uses digraphs but the setup generalizes in a natural way to higher aritys. Using this structure of the constraint satisfaction problem it was shown in [26] the complexity of CSP(A) depends only on a certain set of functions – the polymorphisms of the digraph A.

The exact details of these conditions are described in the Algebraic Dichotomy Conjecture [3] which postulates the necessary and sufficient conditions for the tractability of CSP(A) in terms of polymorphisms (and they proved NP-completeness in the other case). The conjecture was confirmed in various special cases. This set of conditions in terms of polymorphisms is equivalent to defining which constraint languages result in tractable (or not) constraint satisfaction problems. Thus, the problem of determining if a constraint satisfaction problem is tractable is based on the polymorphisms admitted by structure A. As well as this polymorphisms between different digraphs can also reveal information about constraint satisfaction problems. One specific example of where finding a polymorphism between two digraphs is useful involves the Oslak polymorphism. If searching for an Oslak polymorphism between the graphs K3 to K5 the result will return negative. In the paper, The complexity of 3- colouring H-colourable graphs [6] what this has been proven to show is that, given a graph with a promised 3 colouring it is NP-complete to find a 5 colouring of this graph. Thus, revealing the complexity of a constraint satisfaction problem.

Due to the nature of the search space, determining if two digraphs admit a certain polymorphism is computationally challenging. Thus, the focus of this project is on using a variety of methods and techniques to build a specialized tool (PolyFind) capable of efficiently determining if a certain polymorphism exists between two digraphs.

Definitions and examples

Constraint Satisfaction Problem (CSP) is defined as a triple ⟨X, D, C⟩, where:

- X = {x1, …, xn} is a set of variables,
- D = {D1, …, Dn} is a set of the respective domains,
- C = {C1, …, Cq} is a set of constraints,

Where each variable xi can take on values in the nonempty domain Di, every *constraint* Cj∈C is a pair (tj, ρj) where tj is a tuple of variables of length mj, called the *constraint scope*, and ρj is an mj-ary relation on the corresponding domains, called the *constraint relation*. [11]
A conjunctive normal form (CNF) formula on n binary variables is the conjunction of m clauses each of which is the disjunction of one or more literals, where a literal is the occurrence of a variable or its complement . A formula denotes a unique variable Boolean function and each of its clauses corresponds to an implicate of f. Clearly, a function f can be represented by many equivalent CNF formulas. The satisfiability problem (SAT) is concerned with finding an assignment to the arguments of that makes the function equal to 1 or proving that the function is equal to the constant 0. [15]

A *directed graph* (or just digraph) D consists of a non-empty finite set V(D) of elements called vertices and a finite set A(D) of ordered pairs of distinct vertices called arcs. [7]

Throughout the paper Cx and Kx graphs are mentioned, for example C3. Cx represents a cycle digraph of size x, which is a digraph with a set of x nodes where each node has an edge only to the node with a label one great than itself. Or in the case of the last node back to 0. An example of C6 is shown as the left-hand graph in Figure 1. Kx graphs are similar in the respect the consist of a digraph with a set of x nodes. However, for this set of digraphs edges are present between every node, but nodes do not have edges that connect to themselves.

The *tensor product* is defined as follows: For any two digraphs G and H the tensor product of G and H gives the set of vertices that is the combination of every vertex in G with every other vertex in H. For example if vertex 0 was in G and vertex 1 was in H the tensor product of G and H would result in a graph with at least the node (0,1), all the nodes in the output graph have the structure (G.vertex,H.vertex). For this set of vertices, edges only exist if edges between both the G sections of the vertices and the H sections of the vertices have edges between them in the original digraphs. The resulting digraph is the tensor product of graphs G and H. If the tensor product takes in the same digraph as both inputs, then this is said to be squaring the digraph. This pattern is also reflected with higher powers hence when graphs are referred to as being raised to the power or G^n this means the graph is tensor producted by itself n times.

In order to explain a mapping, one must first have two digraphs G1, G2. A mapping from G1 to G2 involves every node in G1 being made equal to a node in G2 through the use of a function. Figure 1 depicts how this is possible. A special type of mapping has the constraint that if an edge exists between two nodes in G1 then the edge must exist between the nodes that these nodes mapped to in G2, in other words the edges are preserved. This can also be described as if the edge {0,1} exists in G1 then the edge {f(0),f(1)} must exist in G2 in order for this special mapping to exist. This special mapping is called a *polymorphism*. Polymorphisms can also have other special constraints. However, most of these special polymorphisms require a graph to be raised to a certain power of itself using the tensor product. An example of a special polymorphism is siggers. The extra constraints of siggers are given by the formulae f(x,x,x,x)=x and f(x,y,y,z) = f(y,x,z,x). What this formula represents is that each variable in these functions can take any value in the domain of nodes in G1. The

function f represents the process of mapping the node inside the function to the node being mapped to. For example, the node (0,1,1,2) in the G1^4 graph (as siggers has a function of arity four), when mapped could be mapped to node 0 in G2 hence f(0,1,1,2)= 0. Now the extra siggers constraint requires that as f(x,y,y,z)=f(y,x,z,x) (x=0,y=1,z=2) the node 1,0,2,0 must also map to the same node, hence f(1,0,2,0) must also equal 0. If this constraint holds for every combination of values that the variables in the function can take and edges are preserved, then the special polymorphism exists between the two digraphs (in this case siggers). If these digraphs are the same, then the digraph is said to have admitted the polymorphism. The polymorphisms mentioned in this paper are defined below.

Let $n \geq 2$ where n is the power the graph is being raised to.

A polymorphism $f : G1^n \rightarrow G2$ is

- a weak near-unanimity polymorphism (wnu n),
  if (for all $x,y \in V$) f(y,x,x,...,x) = f(x,y,x,x,...,x) =⋯= f(x,x,...,x,y);
- a near-unanimity polymorphism (nu n),
  if it is a weak near-unanimity polymorphism of arity $n \geq 3$ and f(x,x,...,x,y) = x;
- a totally symmetric polymorphism (ts n),
  if f(x1,x2,...,xn) = f(y1,y2,...,yn) whenever {x1,...,xn}={y1,...,yn};
- an edge polymorphism (edge n),
  if $n \geq 3$, f(y,y,x,x,...,x) = t(y,x,y,x,x,...,x) = x and
  f(x,x,x,y,x,...,x) = f(x,x,x,x,y,x,...,x) =⋯= f(x,...,x,y) = x;
- a Siggers polymorphism (siggers)
  if n = 4 and f(x,y,y,z) = f(y,x,z,x);
- a Mal'cev polymorphism (malcev),
  if n = 3 and f(y,y,x) = f(x,y,y) = x;
- a semilattice polymorphism (sml),
  if n = 2 and f(x,y) = f(y,x), f(f(x,y),z) = f(x,f(y,z));
- a 2-semilattice polymorphism (2sml),
  if n = 2 and f(x,y) = f(y,x), f(f(x,y),x) = f(x,y);
- an Oslak polymorphism (oslak),
  if n=6 and f(x,x,y,y,y,x) = f(x,y,x,y,x,y) = f(y,x,x,x,y,y)
- a bounded-width polymorphism (omit12),
  s(x,x,x) =x and s(x,x,y) = s(y,x,x) and s(x,x,y) = s(x,y,x) and
  t(x,x,x,x)=x and t(y,x,x,x)=t(x,y,x,x) and t(x,y,x,x)=t(x,x,y,x) and
  t(x,x,y,x) = t(x,x,x,y) and s(x,x,y) = t(x,x,x,y)

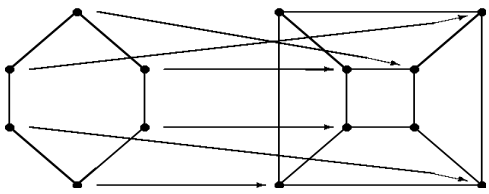

Figure 1. An example of a graph C6 being mapped onto another graph. The arrows represent a function mapping nodes.

II. RELATED WORK

The usefulness of certain polymorphisms has led to a very active research area around this topic. However, the area of research specifically focusing on finding these polymorphisms is very niche, because of this only two such works were found during the research of this project. An applet by Miklós Maróti and a method described in the paper Polymorphisms of Small Digraphs [8].

A. Polymorphisms of Small Digraphs method

The paper Polymorphisms of Small Digraphs describes a method for finding polymorphisms which has the following basic structure. A Pearl script is given a digraph and a polymorphism. The Pearl script will then model the problem of finding if the digraph admits the polymorphism into first order logic. The first order logic problem is then passed to a first order logic model finder called Paradox [24]. Paradox then finds if the first order logic problem is satisfiable or not and returns the result. While the Pearl script was given alongside the paper it did not work as described and thus testing the efficiency of this method proved difficult. The paper [8] does however give some indication on how efficient this method was by providing a list of results. Only one of these results though specifically was given a computation time and that was for graph #235816 shown in Figure 2. This result referred to searching for if graph #235816 admitted the polymorphism nu5, and this took 21 seconds to return a positive answer. Thus, this result is a good comparison to make to other related solutions and PolyFind. The paper also mentions the results of an extensive test that took over one week to compute and involved approximately 3.6 million polymorphism searches. This seemed a more detailed comparison method and therefore the details of the test are given below.

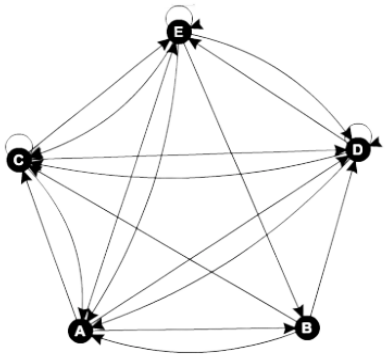

Figure 2. Graph #235816.

The test in paper [8] involves a set of every non-isomorphic digraph from size 2 to 5. Every digraph in this set is tested if it admits a polymorphism in the set: {malcev, sml, nu3, ts4, nu4, 2sml, ts3, edge4, nu5, wnu2, edge5, omit12, wnu5, wnu4, wnu3, siggers}. However, the test is specifically searching for minimal polymorphisms, that is, the lowest polymorphism (shown by Figure 3) that each graph admits. The poset on Figure 3 compares the strength of these polymorphisms. An edge means that the lower polymorphism implies the upper one. In other words, the lower condition on polymorphisms is stronger than the upper one. For example, take the digraph #235816, firstly the digraph is checked if it admits malcev and sml polymorphisms which are at the bottom of Figure 3. If the digraph admits either of these polymorphisms the result is recorded, and this digraph is disregarded. However, if neither of the polymorphisms are admitted by the digraph then, by going up the diagram in Figure 3, a

different polymorphism is searched for (for example the next polymorphism after sml would be nu3). The program terminates once a polymorphism is found to be admitted by the digraph and the result is recorded (in the case of graph #235816 this was the polymorphism nu5). If no satisfiable polymorphisms are found, then the digraph is put in the NONE section of Table 1. Paper [8] gives the times of these computations which are shown in Table 2.

condition	size 2	size 3	size 4	size 5
NONE	0	7	765	155151
malcev	8	29	118	471
nu3	2	63	1572	60056
nu4	0	0	46	8916
nu5	0	0	1	1388
edge4	0	0	0	0
edge5	0	0	0	0
bw	0	0	29	3475
sml	9	90	2178	130870
2sml	0	1	12	1639
ts4	0	0	6	1559
ts3	0	0	0	0
wnu2	0	0	0	0
wnu3	0	0	0	0
wnu4	0	0	0	0
wnu5	0	0	0	0
siggers	0	0	0	0
TOTAL	10	104	3044	291968

Table 1. The number and type of minimal polymorphisms for certain sizes of non-isomorphic digraphs.

Size of graphs	**Time Taken**
2	Always fractions of seconds
3	Always fractions of seconds
4	Always below 2 seconds
5	2.9 million (83%) in < 1 second, 0.6 million (17%) between 1 and 2 seconds, 505 between 2 and 3 seconds, 8 computations over 10 seconds maximum 21s to compute the digraph #235816 admits nu5

Table 2. Computation times for the extensive test in paper [9].

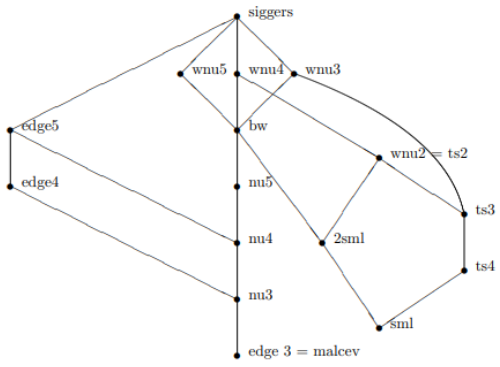

Figure 3. The poset of polymorphisms given by paper [8]

In terms of flexibility the method mentioned in paper [8] did allow for a large number of polymorphism types but did not allow for custom polymorphism inputs therefore this could be an area that PolyFind improves on. As well as this the method only allows for digraphs to be mapped to themselves as opposed to other digraphs, which is another limiting factor. Paper [8] also provides a sample input file to Paradox as shown in Figure 4. This file shows how graph structure, edge preservation, and polymorphism constraints are represented in the Paradox input file. A similar structure could be used as a base for PolyFind.

```
cnf(sml,axiom, t(X,X)=X ).
cnf(sml,axiom, t(X,Y)=t(Y,X) ).
cnf(sml,axiom, t(X,t(Y,Z))=t(t(X,Y),Z) ).
cnf(pr,axiom,( ~gr(X0,X1) | ~gr(X2,X3) | gr(t(X0,X2),t(X1,X3)) )).
cnf(graph,axiom,~gr(n0,n0)).
cnf(graph,axiom,gr(n0,n1)).
cnf(graph,axiom,gr(n1,n0)).
cnf(graph,axiom,gr(n1,n1)).
cnf(elems,axiom,n0!=n1).
cnf(elems,axiom,( X=n0 | X=n1 )).
```

Figure 4. A Paradox input file from [8] asking to determine whether a 2-element digraph admits a semilattice (sml) polymorphism.

B. Applet by Miklós Maróti

Miklós Maróti produced a java applet [21] designed to find if a digraph admitted certain polymorphisms. This applet modelled the problem of whether a given digraph admitted a certain polymorphism into a search problem. A depth first search algorithm implemented in Java was then used to find a solution. The Applet covered very few polymorphism types, just {nu, wnu} and therefore is less flexible than the method mentioned in paper [8]. It does however allow for graphs to be passed into the program via files and not just a string. The applet also has the limitation of only allowing equivalent digraphs to be mapped to each other. Furthermore, no in-depth testing or results were given for this Applet, thankfully the applet was fairly straightforward to set up and run and hence could be tested relative to other methods.

The first comparison involved checking if the graph #235816 admitted the polymorphism nu5. As previously mentioned, this took the method from the paper Polymorphisms of Small Digraphs 21 seconds for a positive result. However, the applet after a few minutes of computing ran out of memory, implying this was a less efficient version. However, one result is not nearly enough to assume the applet is slower in general but unfortunately this was the only concrete result to compare the two methods. Later on, the applet is also compared to PolyFind and the conclusion of this test is in the results section (IV).

III. SOLUTION

In order to produce a solution to the problem of finding if a given digraph admits a certain polymorphism, it is first beneficial to look at the related works. Both Miklós Marótis Applet and the method mentioned in the paper Polymorphisms of Small Digraphs rely on the same core principle. Taking the initial problem and transforming it into some well-known problem before attempting to solve it. Thus, this seems a good starting point to produce PolyFind.

Modelling to a well-known problem

There are a multitude of well-known problems that we could attempt to model this problem to. The aim of this section is to determine which well-known problem would be most suitable for our needs. To start experimentation, two well-known problems were looked at. The constraint satisfaction problem (CSP) and the Boolean satisfiability problem (SAT). In order to test which well-known problem would be more suitable, two python scripts were implemented which transformed very simple polymorphism searches into constraint satisfaction problems and Boolean satisfiability problems. These were then solved by the external programs MINION [22] and MiniSat [23] respectively. It became clear early on that modelling to the constraint satisfaction problem was much more suitable for the tool, taking much less time and memory than the SAT version.

The paper Polymorphisms of small Digraphs mentions another well-known problem that could be modelled to, first order logic. The definition of first order logic is beyond the scope of this paper, but for the purpose of this project first order logic is simply another well-known problem to model too. The paper also mentions a first order model finder called Paradox which was used to solve first order logic problems. Thus, another python script was developed that transformed the problem of finding polymorphisms into first order logic and this problem was then passed to Paradox. The speed and size of problems the first order logic version could deal with were both much greater and therefore this seemed to be a more suitable well-known problem to model to. Paper [8] also mentions that in initial experiments the model builder Mace4 was also used but it was quickly realized the Paradox outperforms Mace4 by an order of magnitude. The paper also concludes that there are other model builders based on different methods, but they do not seem to be able to handle these sorts of tasks efficiently. Thus, modelling to first order logic and then using Paradox to solve the problem seemed the most suitable solution.

PolyFind

PolyFind is a python script which takes an input of two digraphs and a special polymorphism. PolyFind then models the problem of if a polymorphism exists between the two digraphs into first order logic. PolyFind does this by building a set of first order constraints such that the digraphs structure, how they are mapped to each other, edge preservation, special polymorphism constraints and variable domains are all included within the first order logic problem. Once this process is finished the resulting first order logic problem is passed to Paradox which then returns the result back to the python script.

In order to be convenient as possible the final solution has two options for inputting data. These consist of a Graphical User Interface (GUI) and a command line input. The command line option which has the structure *"python polyFind.py -g1 G1 -g2 G2 -poly POLY -polyCustom PC"*. Where G1 is the digraph you are raising to the power, G2 is the digraph you are mapping to, and POLY is the polymorphism you are interested in. PC can be left blank unless a custom polymorphism is required in which case POLY must be set to "custom" and PC is the string that represents the custom polymorphism. For the graph sections of this command you can either type in a list of cycles graphs and K graphs which will be tensor producted together. An example of this would be "C3,K4" or "C7". The other way of passing in a graph is giving a .txt file name such as graph1.txt, this .txt file must be in the same directory as the PolyFind script. The structure of the .txt file must be as follows for the

program to successfully read it: for each node, a list of zeros and ones separated by commas represents for a given node which edges it has. This is repeated for each node and the lists are separated by a semi-colon, for example, C3 would be written down as: 0,1,0;0,0,1;1,0,0. The polymorphism part of the command can be a string with the value of a common polymorphism. The full set of built in polymorphisms is given by the set {malcev, sml, nu3, ts4, nu4, 2sml, ts3, edge4, nu5, wnu2, edge5, omit12, wnu5, wnu4, wnu3, siggers}. Alternatively, if looking for an unusual polymorphism you can define such a polymorphism in the input. The input must be of the structure f(X, …) = f(X..)|X. the ^ symbol represents a new constraint. A few example inputs would be f(X,X,Y) = Y ^ f(X,Y,X) = f(Y,X,Y) or f(X,Y,Z,P) = f(Z,Z,Z,Z) ^ f(X,Y,Z,P) = f(X,X,X,X) ^ f(X,X,X,X) = X.

The GUI option is shown in Figure 5. The GUI is aimed to be as streamlined and intuitive as possible. There is a README.txt included with this program which describes exactly what the GUI and command line are capable of but a brief description in this report is also given. For inputting a graph, you can either type in a graph with the same structure as the command line or give a .txt file name. For the polymorphisms input there is a dropdown box where you are able to select common polymorphisms. There is also an option called custom where you can then type in a polymorphism in the custom input box for example f(X,Y,X)=X. Once all the necessary information has been inputted pressing the find polymorphisms button will generate the Paradox problem file and then call Paradox to attempt to solve the first order logic problem. Any results will then be outputted on the GUI. The inputs have all been labelled for clarity.

PolyFind also has some inbuilt error correction. This specifically looks at the inputs to the program and determines if they are correct. As described above passing in graphs can be done in two ways through the use of a string or a text file. When passing in either of these structures the program checks if the inputs match a regular expression depicting what the inputs must be. The final solution also will show if an entered custom polymorphism constraint is incorrect. An example of an incorrect custom polymorphism constraint would be the constraint f(X,Y)=X ^ f(X,Y,X)=X will throw an error as the function f cannot be a function with an arity of two and three. Errors will be flagged and outputted to inform the user of the mistake.

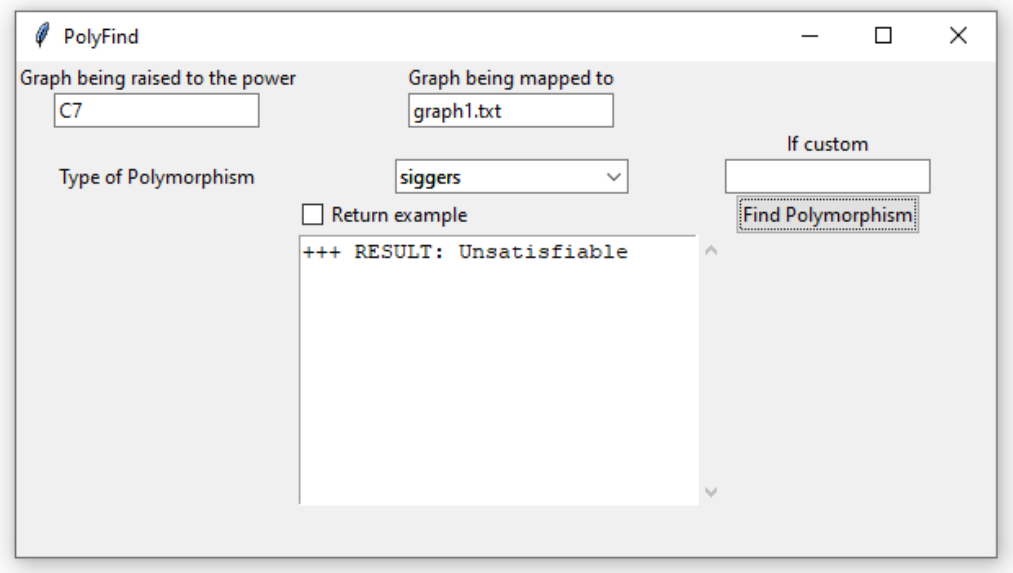

Figure 5. The graphical user interface of PolyFind

External programs

During the production of PolyFind several external programs were used after the modelling process. These programs were MiniSat, MINION and Paradox. This section develops on how the modelling process was adapted to suit the external program. When modelling to MINION initially a variable was used for each node in G1^n and each these varbiles were given domains the size of G2. The variables were then given limitations/constraints based on edge preservation which specified for each variable the values it was allowed to take. This led to a long list of constraints even for relatively simple problems for example, checking for a siggers polymorphism between the digraphs C3 and C3 gave 335KB of constraints. However, it was discovered that MINION has methods of simplifying constraints, especially when a set of variables is under very similar constraints. The constraint in Figure 6 could now be written as just *table([a[0],a[40]],map).* This greatly reduced the size of the constraints, 335KB to 34KB and runtime also dramatically decreased from 16.34 seconds to 0.02 seconds.

```
watched-or({watched-and({eq(a0000a,0),eq(a1111a,1)}),watched-
and({eq(a0000a,0),eq(a1111a,2)}),watched-and({eq(a0000a,1),eq(a1111a,0)}),watched-
and({eq(a0000a,1),eq(a1111a,2)}),watched-and({eq(a0000a,2),eq(a1111a,1)}),watched-
and({eq(a0000a,2),eq(a1111a,0)})})
```

Figure 6. An example of the base versions edge preserving constraint

For the Paradox version optimising for the external program involved taking advantage of the relation structures that first order logic has. Instead of having variables for every node in G1^n both G1 and G2 were described using a relation for example if graph1(0,1) was set to true if an edge was present in G1 from node 0 to node 1. Through this method it was much simpler to then define edge preservation and extra polymorphism constraints. For example edge preservation when looking at a polymorphism of arity 3 could be simply described with cnf(preserves,axiom,(~gr(X0,X1)|~gr(X2,X3)|~gr(X4,X5)|grb(t(X0,X2,X4),t(X1,X3,X5)))). This one line describes how if an edge is present between all the nodes in the original G1 graph then then after the mapping function the edge must exist in G2. This is a much more elegant way of writing edge preservation and may point towards why Paradox seemed much a more suitable for this problem.

Symmetry

The premise behind symmetry is that given a mapping from G1 to G2 that fails, it can be determined that some other specific mappings will fail as well. Therefore, the tool does not have to explore these areas of the search space. While developing the MINION version of the tool, a symmetry was discovered which greatly reduced runtime. Figure 7 gives some examples of this symmetry. Essentially the symmetry involves mappings where nodes can be simply renamed to become equivalent, hence are symmetrical, and thus for each set of these mappings only one mapping needs to be checked. The next step involves making sure MINION does not search these redundant mappings and this was done simply by adding more constraints. Specifically, the first node in G1^n must map to 0 and secondly each node in the mapping process must be at most one more than the maximum of the nodes before it. When

applying this symmetry to the MINION version of PolyFind some problems search times were drastically improved. For example, when looking for a siggers polymorphism between the digraphs C3 and K5 before using symmetry the computation time was 518.73 seconds and but using symmetry to limit the search space meant PolyFind took just 2.57 seconds. List A given by Figure 8 shown the mappings of every combination of three nodes to three nodes. List B given by Figure 9 shows which of the mappings in List A are not equivalent with this symmetry. This demonstrates the drastic reduction in search space that this symmetry causes even a very small problem.
Interestingly when the same symmetry constraints were placed on the Paradox version no improvements in runtime were made. This suggests that Paradox already accounts for such symmetries and hence explains why the Paradox version seems more efficient.

Node Number: 0,1,2,3,4,5,6
Node Mapping: 0,0,1,1,2,2,0

Node Number: 0,1,2,3,4,5,6 = *Node Number:* 0,1,4,5,2,3,6
Node Mapping: 0,0,2,2,1,1,0 = *Node Mapping:* 0,0,2,2,1,1,0

Figure 7. A few isomorphic mappings

```
[(0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 1, 0), (0, 1, 1), (0, 1, 2), (0, 2, 0),
 (0, 2, 1), (0, 2, 2), (1, 0, 0), (1, 0, 1), (1, 0, 2), (1, 1, 0), (1, 1, 1),
 (1, 1, 2), (1, 2, 0), (1, 2, 1), (1, 2, 2), (2, 0, 0), (2, 0, 1), (2, 0, 2),
 (2, 1, 0), (2, 1, 1), (2, 1, 2), (2, 2, 0), (2, 2, 1), (2, 2, 2)]
```

Figure 8. Definition of list A

```
0,0,0
0,0,1
0,1,0
0,1,1
0,1,2
```

Figure 9. Definition of list B

Other alternatives

Through the exploration of different techniques and methods of modelling, a few methods gave results which were worth looking into in greater detail. Generally, these methods gave slower results than the up to date version of PolyFind but for some specific problems they could lead to improvements. These methods were not included in the final version of PolyFind, but they may be useful for future development. Ideally PolyFind would have used these methods in a dynamic way such that it could recognise if a problem would be easier to solve with an alternative method.
One of the problems which was of difficulty for some time was searching for a siggers polymorphism between the digraphs C5 and C3. This problem although relatively small proved quite difficult to crack. An idea to try attempt this problem involves just looking at smaller parts of the problem and seeing if they are solvable. For example, if a section of (C5)^4 could not map to C3 then logically the whole graph could not either. Thus, the problem became

trying to find smaller sections of the C5^4 which were the limiting factor when it came to mapping to C3. This turned out to be an extremely difficult problem to solve even when just looking at an individual problem. C5^4 was split up into different sections which were then attempted to be mapped to C3. Sections that gave yes answers very quickly were ignored and a list of all the sections which were more difficult to solve was produced. Another program was then created to search through these difficult sections, which prioritized looking at smaller sections as these would be faster to compute, abandoning sections if they took too long or gave yes answers. Eventually a section of the graph was found which gave a no answer to the problem of searching for a siggers polymorphism from C5^4 to C3 in just 5.159 seconds. The section of the graph in question involved all nodes f(X,Y,Z,P) where P<=1, X<=4, Y <= 3, Z <= 2. This seemed the most interesting and solvable section for a siggers polymorphism on the set of graphs covered in this paper, and numerous tests were done on this region searching for siggers polymorphisms. Although the results were interesting and is some cases very efficient this method was not used in the final solution. The main reason for this was that positive answers were not guaranteed to be correct as just because a large proportion of the graph maps successfully does not guarantee the whole graph does and thus these results have to be disregarded. However, this definitely seems a reasonable avenue of exploration for further research into this problem for example if a user suspected a certain polymorphism did not exist then this method could verify that result potentially much faster than the default version of PolyFind.

Another idea that was found was taking the graph G1^n and greedily mapping a proportion of the nodes to G2 and then checking if there is a correct mapping from this point. This was mostly focused on the problem of looking for a siggers polymorphism between C7 and K4, which in later results was shown to exist. However, during all experimentation of using different greedy algorithms and different proportions of nodes the problem was either still too difficult to solve or returned negative. Thus, this route of experimentation seemed to be a dead end. There were some problems this method was more efficient at solving than the final version of PolyFind, specifically simple problems where the polymorphism existed, but the search space was large. For example, looking for a siggers polymorphism between the digraphs C10 and C4, but these problems were already solved very quickly by PolyFind and so not relevant for improving the solution.

IV. Results

Throughout this project numerous methods were tried, tested, and, either disregarded or used in the final solution. This section describes what experiments were done, the results of these experiments, and which methods were used in the final version of PolyFind.

To start is the base MINION program. This program although slow was a reasonable break through as testing its results against the results from the paper Polymorphisms of small diagraphs [8] showed the program was giving correct answers for any problems it was capable of answering. However, it struggled with even moderately small problems and was clearly less efficient than any of the related works. A good example of runtime is the problem of looking for a siggers polymorphism between C3 and C3 took 17.67 seconds to return negative as shown in Table 3.

The next development came from adapting the modelling process to suit MINION. After this adaption PolyFind gave much faster results for every test run during the project than the base version of PolyFind. The adapted version of PolyFind also returned results for larger problems the base version would run out of memory for. An example of this is the Table 3 which shows that the problem of looking for a siggers polymorphism between C3 and C3 is

now solved in just 0.02 seconds. The problem of looking for a siggers polymorphism from K6 to C2 which previously was unsolvable in a reasonable time (over 1000 seconds of computation) was also solved in just 0.02 seconds.

Further development came from the previously discussed symmetries of the search problem, which were used to increase the solutions efficiency. Table 3 shows how this method resulted in certain problems such as looking for a siggers polymorphism from the digraph C3 to K5 to be solved much faster. While other simpler problems were simply slowed down by the extra constraints. At this point a few examples were not enough to determine if a new method was more efficient than an old one, as there seemed to be certain problems that some methods were much better at dealing with than others. Thus, a more thorough test to help compare methods was created. This test simply involves the set of graphs {C2,C3…C10,K2,K3…K10} being mapped to itself while searching for a siggers polymorphism (resulting in 324 polymorphism searches). Siggers was chosen as it gives a good range of difficulty for this set of graphs. Figure 10 below shows these results for PolyFind while using the symmetry method.

	C2	C3	C4	C5	C6	C7	C8	C9	C10	K2	K3	K4	K5	K6	K7	K8	K9	K10
C2	0.05	0.045	0.047	0.047	0.05	0.051	0.049	0.122	0.054	0.049	0.046	0.054	0.047	0.052	0.047	0.051	0.053	0.052
C3	0.016	0.019	0.019	0.02	0.022	0.021	0.022	0.027	0.024	0.016	0.018	0.071	2.566	20.13	43.145	49.132	43.784	41.884
C4	0.081	0.082	0.085	0.099	0.1	0.092	0.099	0.105	0.109	0.079	0.092	0.084	0.087	0.1	0.099	0.111	0.118	0.12
C5	0.085	1000.041	0.115	0.115	0.115	0.131	0.131	0.14	0.151	0.078	1000.144	1000.096	1000.07	0.282	0.359	0.34	0.372	0.401
C6	0.552	0.702	0.697	0.989	0.889	0.768	0.735	0.914	1.354	0.491	0.64	0.659	0.714	0.797	0.75	0.887	0.853	0.844
C7	0.691	1000.037	0.441	0.526	0.501	0.6	0.573	0.603	0.636	0.358	1000.038	1000.038	1000.04	0.677	0.707	0.753	0.802	0.837
C8	0.786	1.045	1.041	1.141	1.198	1.267	1.337	1.402	1.481	0.785	1.041	1.139	1.235	1.324	1.415	1.496	1.592	1.665
C9	1.352	1000.039	1.533	2.332	1.711	2.044	1.9	2.273	2.103	1.348	1000.04	1000.039	1000.038	2.603	FAIL	FAIL	FAIL	FAIL
C10	2.823	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	2.813	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
K2	0.046	0.043	0.047	0.048	0.046	0.049	0.052	0.05	0.049	0.048	0.049	0.05	0.046	0.046	0.049	0.049	0.048	0.052
K3	0.017	0.017	0.018	0.019	0.019	0.023	0.024	0.029	0.025	0.016	0.017	0.072	2.541	20.294	43.296	48.867	43.714	42.282
K4	0.101	0.11	0.113	0.117	0.122	0.129	0.132	0.138	0.144	0.101	0.114	12.441	1000.038	1000.038	1000.038	1000.037	1000.037	1000.039
K5	0.677	0.751	0.725	0.756	0.777	0.804	0.829	0.857	0.878	0.669	0.751	1000.04	1000.051	1000.053	1000.074	1000.075	1000.047	1000.042
K6	3.327	3.732	3.745	3.703	3.791	3.888	4.009	4.107	4.234	3.311	3.743	1000.043	1000.049	1000.047	1000.053	1000.08	1000.05	1000.046
K7	12.412	13.791	13.192	13.529	14.072	FAIL	FAIL	FAIL	FAIL	12.347	13.988	1000.04	1000.08	1000.059	FAIL	FAIL	FAIL	FAIL
K8	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
K9	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
K10	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL

Figure 10. The results of all 324 siggers tests produced by the symmetry version of PolyFind. The vertical axis is mapped to the horizontal one. Red represents a polymorphism was not found between the two digraphs. Green represents a polymorphism was found between the two digraphs. Purple represents PolyFind could not return an answer within 1000 seconds. Yellow represents that Paradox did not have enough memory to attempt the first order logic problem.

Of the 324 polymorphism searches the version of PolyFind focused on symmetry managed to give yes or no answers to 207 problems. Considering lots of these problems are much larger than covered in the paper Polymorphisms of small diagraphs this seems a competitive set of results. However, there are still some small problems that seem to be too difficult to solve, the smallest being finding a siggers polymorphism between the digraph C5 to C3. This method also fails when dealing with any larger digraphs as it simply runs out of memory trying to solve the problem. In these cases, the resulting problem may be very simple to solve but is just too large for the computer to handle. This suggests that there are definitely ways to improve the solution.

Figure 11 looks at the results of the program that just looked at part of the graph (to be precise the nodes that satisfy f(A,B,C,D) where A<=4,B<=3,C<=2,D<=1. The 324 tests of finding siggers from above was also run with this method giving the results shown below.

	C2	C3	C4	C5	C6	C7	C8	C9	C10	K2	K3	K4	K5	K6	K7	K8	K9	K10
C2	0.052	0.055	0.05	0.053	0.053	0.054	0.053	0.058	0.058	0.051	0.057	0.053	0.053	0.052	0.055	0.052	0.053	0.053
C3	0.011	0.013	0.016	0.013	0.014	0.014	0.017	0.017	0.017	0.011	0.014	0.025	10.239	829.44	1000.16	1000.187	1000.05	1000.042
C4	0.059	0.061	0.06	0.074	0.067	0.07	0.07	0.074	0.074	0.061	0.075	0.064	0.065	0.06	0.08	0.072	0.075	0.075
C5	0.029	5.159	0.04	0.04	0.043	0.047	0.047	0.053	0.056	0.029	5.137	0.088	0.083	0.085	0.09	0.096	0.098	0.101
C6	0.089	0.105	0.104	0.109	0.115	0.121	0.126	0.133	0.139	0.091	0.098	0.102	0.109	0.117	0.125	0.13	0.136	0.142
C7	0.073	1000.042	0.091	0.136	0.109	0.129	0.128	0.138	0.147	0.073	1000.041	0.144	0.153	0.167	0.175	0.192	0.201	0.208
C8	0.157	0.186	0.195	0.21	0.225	0.24	0.254	0.269	0.282	0.159	0.187	0.203	0.219	0.23	0.251	0.264	0.283	0.296
C9	0.161	1000.042	0.203	1000.123	0.81	0.856	0.677	0.884	1.054	0.537	1000.124	0.923	0.832	0.941	0.954	1.103	1.324	1.061
C10	0.69	2.098	1.21	1.065	1.995	1.18	1.308	2.036	1.563	0.729	0.835	1.033	1.172	1.399	1.231	1.459	1.468	1.829
K2	0.146	0.138	0.117	0.123	0.179	0.157	0.151	0.224	0.174	0.204	0.148	0.13	0.161	0.165	0.194	0.143	0.174	0.181
K3	0.038	0.019	0.035	0.028	0.027	0.034	0.031	0.031	0.028	0.021	0.038	0.075	25.825	1000.043	1000.044	1000.046	1000.044	1000.041
K4	0.029	0.033	0.033	0.035	0.037	0.039	0.039	0.042	0.045	0.026	0.032	0.313	457.698	1000.046	1000.046	1000.042	1000.043	1000.042
K5	0.086	0.107	0.1	0.105	0.108	0.119	0.122	0.128	0.132	0.084	0.106	7.149	1000.042	1000.044	1000.043	1000.045	1000.043	1000.043
K6	0.27	0.398	0.294	0.322	0.325	0.341	0.363	0.38	0.393	0.263	0.397	153.571	1000.053	1000.054	1000.051	1000.061	1000.051	1000.044
K7	0.67	1.1	0.736	0.8	0.816	0.861	0.907	0.956	1.006	0.668	1.099	1000.167	1000.044	1000.043	1000.052	1000.067	1000.049	1000.043
K8	1000.043	1000.062	1000.047	1000.044	1000.044	1000.046	1000.045	1000.053	1000.083	1000.043	1000.057	1000.042	1000.045	1000.047	1000.05	1000.052	1000.054	1000.181
K9	1000.268	1000.173	1000.275	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
K10	1000.268	1000.173	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL

Figure 11. the results of all 324 siggers test produced by just looking at part of the problem (or a segment). See Figure 10 for colour keycode.

Of the 324 tests the PolyFind version which involved looking at segments returned answers for 232 of the problems. This is 25 problems better than just using symmetry. These newly solved problems included one of the small difficult problems the symmetry version struggled with. The downside of this method however is returned yes answers are not definitely true which arguably means the method only returned 103 answers. So how useful is this method. Well it did solve some problems faster than just looking at symmetry and give a good indication of whether some problems should have a siggers polymorphism. However, it was very computationally expensive to find this one interesting region and this region is specifically focused at just siggers polymorphisms. Doing this for every individual polymorphism request is unrealistic. If PolyFind was designed to find a single polymorphism this would be an interesting avenue to explore. However, the tool being developed needs to be able to deal with a large variety of polymorphism types. Therefore, PolyFind searching for interesting parts of every problem it attempts to solve is to computationally expensive to be worth investing the time for an unknown polymorphism. On the other hand, for common polymorphisms it may be worth investing the time to find interesting parts of the graphs but due to the nature of only negative answers being absolutely correct this was not a priority for the development of PolyFind. It is worth noting however that using later results, all the returned results in this test were in fact correct except for one case, which was searching for a siggers polymorphism between the digraphs C5 to K4 which returned positive when it should have returned negative.
The next big breakthrough came from using Paradox as the external program instead of MINION. After producing a first order logic version of the python script, the same 324 tests mentioned before were attempted by Paradox and the results are shown below.

	C2	C3	C4	C5	C6	C7	C8	C9	C10	K2	K3	K4	K5	K6	K7	K8	K9	K10
C2	0.029532	0.024967	0.069795	0.105021	0.228372	0.555128	1.23968	2.742011	4.940281	0.029324	0.023163	0.034626	0.056016	0.096171	0.175445	0.320602	0.576068	0.936647
C3	0.027274	0.027305	0.036393	0.1081	0.13219	0.25034	0.434837	0.840019	1.355976	0.025127	0.025953	0.04634	0.073882	0.162474	0.336242	0.664812	0.854931	1.545706
C4	0.044186	0.060353	0.02344	0.112882	0.258598	0.624827	1.593797	5.914914	11.50744	0.043645	0.047329	0.040321	0.0685	0.153714	0.260062	0.424645	0.736548	1.218639
C5	0.083694	0.083183	0.074482	0.07359	0.166369	0.317828	0.593586	1.012384	1.711353	0.07867	0.081609	0.079437	0.102242	0.204492	0.352872	0.608689	1.011485	1.355936
C6	0.237146	0.403375	1.625718	0.399597	0.45722	1.035522	3.563179	6.979464	25.47533	0.222453	0.411935	0.378366	0.372568	0.169646	0.300208	0.512986	0.902584	1.432554
C7	0.420122	0.435156	0.408964	0.39601	0.394294	0.407133	0.734749	1.293662	2.116456	0.422723	0.430266	2.983513	1.262985	1.410442	0.315146	0.567264	0.982811	1.509515
C8	1.026124	3.326719	21.51252	4.565398	4.05845	5.341347	3.184561	6.394406	25.72096	1.022188	3.404003	3.956536	4.436198	5.966555	5.075194	0.607168	1.016527	1.608009
C9	1.734546	1.808218	1.775446	1.658733	1.573079	1.556217	1.583389	1.48426	2.577579	1.662945	1.832899	5.622103	32.06613	13.69821	13.96162	11.97916	1.0793	1.745307
C10	3.544396	23.35859	105.3361	16.70916	21.68076	28.1372	19.99616	20.05112	21.84158	3.705645	23.8946	23.49912	31.15449	36.56589	37.21769	39.60411	36.4052	1.831407
K2	0.026228	0.026623	0.064349	0.091582	0.220442	0.553864	1.133872	2.586828	4.94504	0.023354	0.024499	0.050009	0.057268	0.095725	0.176349	0.329165	0.572573	0.897654
K3	0.031013	0.023911	0.03705	0.066929	0.131124	0.260986	0.441024	0.808169	1.427118	0.025974	0.024733	0.0413	0.075233	0.160607	0.344703	0.686441	0.847783	1.525925
K4	0.04515	0.041979	0.042507	0.082958	0.16453	0.339277	0.64672	1.135739	2.011745	0.042872	0.043229	0.037798	0.108624	0.207498	0.401516	0.783034	1.486233	2.419497
K5	0.111672	0.107607	0.103391	0.098174	0.233826	0.421096	0.891956	1.521387	2.525236	0.111692	0.106107	0.093066	0.079423	0.230887	0.397092	0.908642	1.604173	2.535672
K6	0.349519	0.332868	0.294389	0.29806	0.304832	0.593679	1.147256	2.031629	3.390315	0.353205	0.331504	0.28731	0.249931	0.220145	0.547388	0.978503	1.720074	2.378893
K7	0.921948	0.864758	0.825828	0.804539	0.801516	0.797043	1.455123	2.562696	4.428716	0.885793	0.883091	0.801617	0.688824	0.591525	0.470296	1.098731	1.978073	3.245593
K8	2.186311	2.107791	2.073824	2.026389	1.980823	1.995329	1.889058	3.223275	5.559861	2.194212	2.186314	1.977148	1.783018	1.591689	1.322162	1.013266	2.234497	3.423089
K9	4.714089	4.697967	4.547447	4.612829	4.396289	4.364875	4.32418	4.230271	7.613969	4.698242	4.667671	4.451108	4.244982	3.657839	3.192929	2.684498	1.995491	4.116832
K10	10.01067	10.23917	9.538491	9.586667	9.491902	9.419024	9.145802	9.04076	8.94644	10.0638	9.923103	9.342495	8.771562	8.145328	7.240705	6.472048	5.186288	3.869511

Figure 12. the results for all 324 siggers tests while using Paradox as an external modelling program. See Figure 10 for colour keycode.

As shown by Figure 12 the Paradox version returns results for every single one of the test cases. Even if this method were slow, its consistency would make it a contender for being used in the final solution of PolyFind. However, the efficiency of this method is easily comparable to the other methods currently implemented and is faster for the majority of problems. It may be interesting for the reader to note the pattern emerging for odd cycles mapping to K graphs produces a step like pattern. This pattern does not continue in either direction after these 3 values.

An explanation for why the Paradox version seems so much more effective is shown in Figure 13. What Figure 13 shows is the computational path from the problem to the solution. The blue line represents how much of the computational process (modelling) designed during this project. The boxes represent external programs and black singular lines running horizontally represent the modelling and solving done by these programs. Now naturally the modelling done by this project will be the least efficient part of the process when compared with programs developed by experienced computer scientists/mathematicians. Now as Paradox is at a higher level (more abstract) than MINION and also uses MiniSat after its computation it puts less burden on this project to model the problem. This idea is verified by the fact that symmetries which improved the MINION line of computation had no impact on the Paradox line suggesting that these symmetries had already been accounted for.

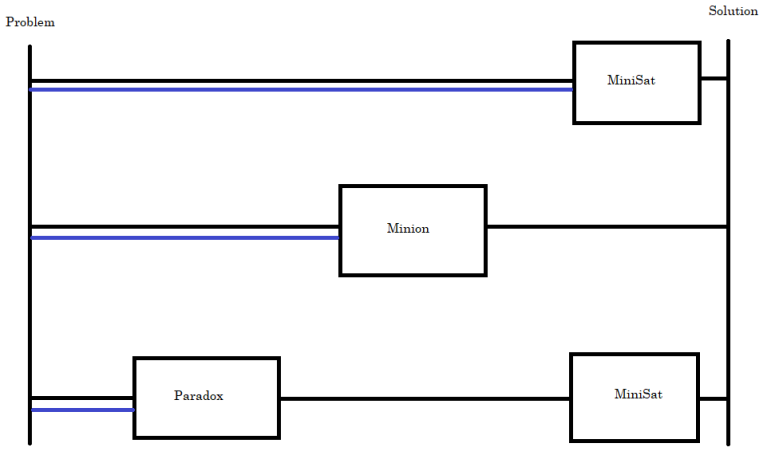

Figure 13. The computational pathways of different solutions

The next step involved comparing the tried and tested methods in order to determine the version that would be used in PolyFind. Throughout this project it was found that judging a methods efficiency comes down to two factors. The difficulty of problems a method can tackle, and the time taken for a method to solve a given problem. As the time taken for a problem to be solved is mostly irrelevant if under ten seconds the comparison method primarily focuses on which method is capable of dealing with the largest variety of problems. Table 3 shows some of the cut off points of each method. This table clearly shows how the Paradox and segment version were the preferable options due to their ability to solve the majority of problems covered by the 324 siggers test also shown in Figure 11 and Figure 12. However, due to the limitations of just looking at segments of a problem the selected method for the final version of PolyFind was the Paradox method. Now that the method for the final solution has been selected the next set of experiments compare the results of the final version of PolyFind to the two other methods mentioned in the related works (Miklós Marótis applet and the Perl Script mentioned in the paper Polymorphisms of small digraphs) both for correctness and efficiency.

Pair of digraphs	Base version	Clean version	Symmetry version	Segment version	Paradox version
C3->C3	17.67	0.02	0.02	0.01	0.03
K6 -> C2	Over time	0.02	3.33	0.27	0.35
C3 -> K5	Over time	518.73	2.57	10.24	0.074
C5 ->C3	Over time	Over time	Over time	5.159	0.083
C7 ->K4	Over time	Over time	Over time	in: 0.144	2.984
C10->C4	Over time	Over time	Over time	in: 1.21	105.3361

Table 3. Important problems that gave certain methods difficulty. Over time represents over 1000 seconds of computation with no result returned. Results with in: before them are inconclusive and not necessarily correct. The in: results in this table however are correct.

The first set of tests involve comparing PolyFind to the applet Miklós Maróti produced. As the applet could only deal with a very limited number of polymorphism types, the test just includes the polymorphism nu. Both programs were given the task of computing for all non-isomorphic digraphs of size 4 which admitted nu4. This test was chosen as there were enough problems (3044) to give a good indication of the faster method as well as providing problems which were a suitable level of difficulty. Both Miklós Marótis applet and PolyFind were given this set of tests and they returned the exact same results. The list of results is too large to show in this report but 1736 of the graphs returned positive answers and 1308 returned negative answers for both methods. While this does not ensure correctness, it does imply that PolyFind was working as intended. Both of the methods were timed during this computation and Miklós Marótis applet managed to solve all 3044 problems in 662.46 seconds. Whereas PolyFind solved these problems in just 118.98 seconds, approximately 20% of the time. This gives PolyFind a good indication of being the faster method. This claim was supported by another result which involved looking at if digraph #235816 admitted the polymorphism nu5. The applet after a few minutes of computation ran out of memory whereas PolyFind returns a positive response in under two seconds. A further test was run to verify PolyFind as the more efficient option which involved giving both programs random size 5 digraphs and checking if

they admitted a nu5 polymorphism. The applet would occasionally run out of memory and not return a result for this set of problems whereas PolyFind always returned as result in under 2 seconds. All these results together imply that PolyFind has a significant advantage when searching for polymorphisms between digraph than Miklós Marótis applet.

The next section discusses the comparison of the final version of PolyFind to the method used in the paper Polymorphisms of small digraphs [8]. The paper does mention one particular result which was it took 21 seconds to compute that the digraph #235816 shown in Table 2 admits nu5. As previously mentioned, using PolyFind to solve this problem took under two seconds which is a resounding improvement. Although this points towards PolyFind being faster than the method mentioned in paper [8] one result as described earlier when comparing different methods which search for polymorphisms is not enough. Thus, an experiment was run to replicate the test done in the paper Polymorphisms of Small Digraphs which is described in the related works section (II) of this paper. The time taken to get the results of this test is given by Table 4. PolyFind successfully replicated this test and returned an identical table to that of Table 1 for the results of this experiment. Although this does not prove correctness it greatly implies it. The time taken for PolyFind to solve this problem is shown in Table 2.

Size of graphs	**Time Taken**
2	Always fractions of seconds
3	Always fractions of seconds
4	Always fractions of seconds
5	3.6 million (99%) in < 1 second, 44 between 1 and 2 seconds, 1 > 2 seconds which was the maximum time of 2.10 seconds

Table 4. Time taken for PolyFind to complete the extensive test of 3.6 million polymorphism searches mentioned in paper [8].

When comparing Table 4 and Table 2 for this large set of 3.6 million polymorphism tests, it was found that PolyFind outperforms the method found in paper [8]. The maximum time of all the tests is reduced by 90% when using PolyFind and the number of problems that took over a second was reduced from approximately 0.6 million to just 45.
The result of comparing PolyFind to the two related works shows that by a significant margin PolyFind is an improvement on the related works.

V. EVALUATION

In this Section an evaluation of the solution is given by using the results presented in the previous section. The strengths and weakness of the solution are discussed and, the approach to this project is criticized.

PolyFind Strengths

The Project covers a wide variety of modelling techniques and methods for attempting to solve the problem of finding if two digraphs admit a polymorphism. It has been demonstrated

that the produced tool PolyFind is more efficient than any of the current alternatives found, solving the hardest problem mentioned in the paper Polymorphisms of Small Digraphs in 90% less time. PolyFind also has a high consistency as shown by the test involving 3.6 million polymorphism searches, where PolyFind had a maximum time of just 2.10 seconds. As well as this PolyFind has a large amount of flexibility, this is shown by being able to have custom polymorphisms as inputs. As well as being able to take in digraphs in a multitude of ways all while being simple and intuitive to use. The produced tool also includes built in error correction to help correct mistakes with inputs. Overall PolyFind is a robust, flexible tool which can competitively find if a certain polymorphism exists between two digraphs.

PolyFind Limitations

There are several limitations of the Produced tool PolyFind in this report. One such limitation for the produced tool PolyFind is that it is undynamic. Throughout this paper it has been shown that certain methods are better at solving specific problems. For example, take the problem of looking for a siggers polymorphism between C20^4 to C2 this has a very large but relatively simple search space and yet Paradox does not have enough memory to deal with search spaces this large. Even if it were the case that the first mapping looked at was successful Paradox would not be able to solve the problem. In such a case a partial greedy algorithm would be much more suitable. Another example of this involves the segment method where large problems which returned negative answers could be shrunk to trivial sizes. Perhaps this could have been done at the start of every search just to quickly check if the problem was simple and returned negative. However, the undynamic nature of PolyFind does not allow it to adapt to problems hindering its flexibility.

Another limitation is that this paper does not particularly look at how much memory a certain implementation takes, because of this PolyFind has not been optimised to use the least amount of memory possible. As memory is sometimes the limiting factor for PolyFind this is a definite limitation. On the other hand optimising the time taken for PolyFind to solve a given problem often meant increasing memory efficiency as well, resulting in the final version of PolyFind to be somewhat memory efficient but very little testing was done in this area to confirm this.

Finally, PolyFind does not take full advantage of the symmetry of the search space. During this project, a large segment of time was set on finding different versions of symmetries and although symmetries not yet mentioned in the paper were found these relied on specific graph structures and so were not used in the final version of PolyFind. Further research is appropriate for fully exploring the area of symmetries that the problem of searching for polymorphisms between two digraphs represents.

Approach

Our approach initially consisted of building a very simple version of PolyFind and then using an agile approach to improve it. This involved taking the base tool and then using an iterative cycle of trying new methods/routes to solve the problem, which would be implemented, tested, and then either disregarded or added to the tool. A great portion of time spent on this project was on this recursive stage. Eventually the tool was competitive enough to compare to the related works. On comparing with the methods from the related works, it was found some of the conclusions that were drawn from experimentation, were confirmed by the related works (for example that Paradox is a very suitable external program to solve this problem). If this project were to be run again, instead of starting with a basic crudely

implemented version of PolyFind it would be more efficient to start with the fastest of the related works and then, recursively improve this already highly optimised program by using any other modelling methods and resources covered by this paper. This would result on more time focused on new ideas and improvements rather than replicating the improvements shown in the related works. The final program PolyFind was the most competitive out of all the current options but a lot of time was wasted by starting from the ground up. This would have given more time on improving PolyFind which would have been beneficial as there seems to be many more viable approaches to solving this problem. One such approach includes genetic algorithms which although not always correct may have been much better at handling large search spaces.

VI. Conclusion

In terms of fulfilling the aims, the project was an overall success. PolyFind, the produced tool, is a flexible and robust software tool which can search for special polymorphisms between digraphs. In order to do this a variety of modelling methods and techniques were implemented, tested and then either disregarded or used in PolyFind. PolyFind was also compared to other current alternatives and for the set of tests covered in this paper PolyFind was shown to be more efficient than any of these alternatives.

The project also gives an indication of a suitable direction for future work. A dynamic version of PolyFind which could adapt its method to the problem it was attempting to solve would be beneficial, as it has been shown certain algorithms and methods fair better solving certain problems. Methods which were experimented with but not used could also be looked into in further depth. For example, although the segment method was eventually not used in PolyFind if it were possible to prove that certain areas of the graph would always return the correct result this would greatly decrease the search space when looking for polymorphisms. Another path of investigation could have focused on memory efficiency as larger and larger problems are being tackled, this could become a greater issue.

References

[1] Andrei A. Bulatov, Andrei A. Krokhin, and Peter Jeavons. Constraint satisfaction problems and finite algebras. In Automata, languages and programming (Geneva, 2000), volume 1853 of Lecture Notes in Comput. Sci., pages 272–282. Springer, Berlin, 2000.

[2] Andrei Bulatov, Peter Jeavons, and Andrei Krokhin. Classifying the complexity of constraints using finite algebras. SIAM J. Comput., 34(3):720–742 (electronic), 2005.

[3] Andrei A. Bulatov. Tractable conservative constraint satisfaction problems. Logic in Computer Science, Symposium on, 0:321, 2003.

[4] Andrei A. Bulatov. A dichotomy theorem for constraint satisfaction problems on a 3-element set. J. ACM, 53(1):66–120 (electronic), 2006.

[5] Andrei A. Bulatov and Matthew Valeriote. Recent results on the algebraic approach to the CSP. In Nadia Creignou, Phokion G. Kolaitis, and Heribert Vollmer, editors, Complexity of Constraints, volume 5250 of Lecture Notes in Computer Science, pages 68–92. Springer, 2008.

[6] A. Krokhin and J. Opral. The complexity of 3- colouring H-colourable graphs. In Proceedings of the 60th Annual IEEE Symposium on Foundations of Computer Science (FOCS'19), 2019, arXiv:1904.03214.

[7] Bang-Jensen, J. and Gutin, G., 2009. Digraphs. 2nd ed. Dordrecht: Springer, p.2.

[8] Barto, Libor & Stanovský, David. (2010). Polymorphisms of small digraphs. Novi Sad J. Math.. 40. 95-109.

[9] COHEN, D., JEAVONS, P., AND KOUBARAKIS, M. 1997. Tractable disjunctive constraints. In Proceedings of the 3rd International Conference on Constraint Programming—CP'97 (Linz, Austria, October). Lecture Notes in Computer Science, vol. 1330. Springer-Verlag, New York, pp. 478–490.

[10] Cristian Frˇasinaru. Omnics. http://omnics.sourceforge.net.

[11] D. Zhuk. 2018. A modification of the CSP algorithm for infinite languages. CoRR abs/1803.07465 (2018)

[12] Francois Laburthe and Narendra Jussien. Choco. http://choco.sourceforge.net.

[13] Frasinaru, Cristian. (2007). Basic Techniques for Creating an Efficient CSP Solver. Scientific Annals of Computer Science. 17.

[14] Ilog. Ilog solver. http://www.ilog.com.

[15] J. P. Marques-Silva and K. A. Sakallah, "Boolean satisfiability in electronic design automation", *Proc. Design Automation Conf.*, pp. 675-680, 2000-Jun.

[16] Koalog. Koalog constraint solver tutorial. http://www.koalog.com.

[17] L. Barto and M. Kozik, Constraint satisfaction problems of bounded width, in Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science (FOCS'09), 2009, pp. 595–603.

[18] Libor Barto, Marcin Kozik, and Todd Niven. The CSP dichotomy holds for digraphs with no sources and no sinks (a positive answer to a conjecture of Bang-Jensen and Hell). SIAM J. Comput., 38(5):1782–1802, 2008/09.

[19] Libor Barto and Marcin Kozik. Constraint satisfaction problems of bounded width. In FOCS '09:Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science, pages 595–603, 2009

[20] Libor Barto, Andrei A. Krokhin., and Ross Willard. Polymorphisms, and How to Use Them. In The Constraint Satisfaction Problem: Complexity and Approximability [Result of a Dagstuhl Seminar], pages 1–44, 2017.

[21] Miklós Marótis applet. https://github.com/mmaroti/shared

[22] MINION constraint satisfaction solver. https://constraintmodelling.org/minion/

[23] MiniSat Boolean satisfiability solver. http://minisat.se/

[24] Paradox first order model finder. https://github.com/nick8325/equinox

[25] Pavol Hell and Mark Siggers. Semilattice polymorphisms and chordal graphs. European J. Combin., 36:694–706, 2014. doi:10.1016/j.ejc.2013.10.007.

[26] Peter Jeavons, David Cohen, and Marc Gyssens. Closure properties of constraints. J. ACM, 44(4):527–548, 1997.