Predicting the runtime of combinatorial problems CS6780: Machine Learning Project Final Report

Eoin O'Mahony

December 16, 2010

Abstract

Solving combinatorial problems often requires expert knowledge. These problems are prevalent in many aspects of modern life and being able to model and solve these problems quickly is important. Many different modalities exist to solve these problems; such as Mixed Integer Programming, Constraint Programming and Satisfiability. These approaches have greatly varying performance. Choosing a solution method requires years of experience in the domain. The aim of this work is to automate part of this process by developing machine learning methods that will predict the run time of different solvers.

1 Introduction

The goal of this work is to aid users in solving combinatorial problems quickly. Combinatorial problems occur in many areas of computer science, operations research and mathematics. Many different approaches exist for solving these problems, the three that this work focuses on; Constraint Programming, Satisfiability and Mixed Integer Programming on are described in detail below. These methods have complimentary performance on many problems and much research has been carried out investigating their performance on different problem classes.

For novice users, solving combinatorial problems can be a challenge. In systems such as Numberjack, Hebrard et al. [1] users can choose between different paradigms. It often difficult for non expert users to gauge which is the best approach for a given problem. This work is concerned with developing a system for Numberjack that can help automate this task by predicting the runtime of these solvers on a given problem.

The remainder of this section will outline the three different methods used to solve these problems. Section [2] outlines related work in the area of machine learning applied to predicting the runtime of combinatorial problems. Section [3] deals with the gathering of the dataset for this work while Section [4] details the features used for these problem. Finally Sections [5, 6] deal with the machine learning methods used and the results that they yielded.

1.1 Constraint Satisfaction Problems

A *constraint satisfaction problem* (CSP) is defined by a finite set of variables, each associated with a domain of possible values that the variable can be assigned, and a set of constraints that define the set of allowed assignments of values to the variables. The *arity* of a constraint is the number of variables it constrains. Given a CSP the computational task is normally to find an assignment to the variables that satisfies the constraints, which we refer to as a *solution*. To find a solution to a CSP we can either use *systematic search*, usually based on backtracking, or *incomplete search*, usually based on a local search procedure.

1.2 Mixed Integer Programming

Mixed Integer Programming is another paradigm of optimisation. It is concerned with finding a solution to a system of linear equations such that an objective function is maximal or minimal. Unlike simple linear programming where variables have continuous domains in Mixed Integer Programming (MIP) variables can be constrained to have integer values. The field of linear programming started in 1947 with G. B. Dantzig developing the *simplex method* for solving linear programming formulations of air force planning problems. It has grown from this to become an integral part of modern operations research. Although linear programming is tractable Mixed Integer Programming is intractable. Methods for solving MIPs include branch and bound search using the linear relaxation as a lower bound, problem decomposition and advanced methods such as cuts.

1.3 SAT

Satisfiability (SAT) is the process of determining if a set of variables can be given boolean true or false values such that a given expression using these variables evaluates to true. In 1971 Stephen Cook showed that SAT problems are NP-complete. SAT problems are usually specified as a conjunction of disjunctive clauses. a clause if the clause evaluates to true under the assignment. SAT solvers use method such as local or exhaustive search with advanced features such as clause learning to solve problems quickly.

2 Previous work

Previous work in this area has centered around the creation of algorithm portfolios and automatic tuning of solvers. Algorithm portfolios aim to exploit the strengths of different algorithms by choosing between them on an instance to instance basis.

The SAT portfolio solver SATZilla Xu et. al. [8] has enjoyed enormous success, winning five medals in both the 2007 and 2009 SAT Competition. They use a ridge regression approach to predict the runtime of different SAT solvers on a given instance.

O'Mahony et al. [5] developed a similar approach for Constraint Problems; developing cpHydra. Given a problem instance CpHydra computes a schedule of solvers to maximise the chance of solving the problem within a given time limit. Like SATZilla, cpHydra has enjoyed success in competitions, winning the 2008 Constraint Solver Competition.

Hutter et al. [2] have developed a system to automatically tune Mixed Integer Programming solvers. Their method involves selecting the best set of parameters controlling the MIP solver on an instances to instance to basis depending on problem features parsed from the problem.

The main contribution of this work is the novel idea of mining features from different representations of the problem. While only SATZilla is concerned with predicting actual runtime this work aims to greatly expand on this by predicting the runtime of different solving paradigms.

3 Dataset

Previous work relating to constraint satisfaction problems [5] used data from the International Constraint Solver competition. This competition runs on a set of instances that are described in the XML based XCSP language [6]. Given the diverse problem instances written in this format publicly available as benchmarks this format was chosen as the format for the problem instances used. A wide range of benchmark instances were gathered for the construction of the dataset.

Creating the dataset proved to be a challenge. Problems were encountered with the XCSP language and the difficulty involved in parsing it. This was mainly due to many constraints having representations in an arbitrary format as opposed to pure XML. Initially a parser was constructed from scratch but this emerged to be too time consuming. Alternatively a version of the solver Mistral was modified to allow extraction of the parsed XCSP model.

Another difficulty in building the dataset emerged from encoding problems into SAT and MIP models. Despite all three paradigms being reducible to each other, as they are all NP-Complete, efficient decompositions for some constraints are difficult to find. This is particularly true for constraints that are highly non linear when decomposing to MIP. XCSP problem instances are broken up into two classes of constraints; extensional and intensional. Intensional constraints are constraints that constraint variables by specifying a relation between, for example $x_1 < x_2$ is an intensional constraint. Extensional constraints express the set of allowed or disallowed values that sets of variables can take. In this work only problems specified in intension are used in the dataset.

To gather features for each problem it is necessary to first load the problem into a Numberjack model, this model is subsequently decomposed into its SAT and MIP representations. Features are then parsed from the three representations.

To gather the runtime data all three solvers Mistral, MiniSat and SCIP were run on each problem. The solvers were run with a time limit of fifteen minutes. This was to ensure that a decent sized dataset could be gathered. The solvers were run on a cluster with each process running on a Quad Core 2.6Ghz XEON with 16Gb of RAM available. It was crucial to have the processes run on independent hardware to gather reliable runtime data. The final dataset gathered has runtime on all solvers and features for 1042 problem instances ranging over a wide variety of problem classes.

While developing this Parser the author obtained data from previous work [5], described in Section [2]. The results for different machine learning approaches and features selection algorithms on this data are described in detail in [Section 5].

4 Problem Features

Initially it was assumed that features similar to those in previous work would be effective indicators of the runtime of problems. Unfortunately this turned out to not be the case as will be explained in [Section 5]. To remedy this the author started with a stripped down set of simple features.

These features come from three separate sets. The first set of features are gathered by a static analysis of the constraint satisfaction problem representation of an instance. This yields features such as

- **# Constraints** :- the number of constraints found in the problem
- **# Variables** :- the number of variables in the problem
- **# Expressions** :- the number of expressions in the problem, this differs from the number of constraints as a large number of constraints can be enclosed in a single expression
- **Average constraint arity** :- the average number of variables involved in each constraint
- **Average domain size** :- the average number of values in variable domains

The second set of features comes from decomposing the problem into its SAT representation. Thins involves decomposing all variables into a series of true false literals and doing the same for the constraints used in the problem. The decomposition methods of Tamura et. al [7] were used. Although the decomposition methods used are state of the art on some problems the memory usage was much higher than the original problem. Some constraints require quadratic growth in the domains of the variables they constrain. These instances were handled by setting a memory limit which if exceeded resulted would result in a sparse feature vector. Some of the features gathered from the sat instances are shown below:

- **# Clauses** :- the number of SAT clauses in the decomposed problem

- **# Literals** :- the number of SAT literals in the problem
- **Average size of clause** :- the average number of literals found in a clause
- **# Clauses / # Literals** :- the ration of clauses to literals in the problem

The final set of features comes from encoding the CSP as a Mixed Integer Programming problem. The decomposition methods used are a compilation of well known methods in the literature and a number of novel decomposition methods. As with the SAT decomposition some problems are simply too large to decompose to linear constraints. This is a result of highly discretised constraints requiring decomposition of variables to weighted sums of binary variables and then having a super quadratic number of linear constraints over these variables. Again these problem instances were identified and also contributed a sparse feature vector. The goal of the MIP features is to identify those problems whose MIP formulation is ideal for a MIP solver. These features relate to things such as density of the LP matrix and information about density of individual constraints. Some examples of these features are listed below:

- **# Linear Constriants** :- the number of linear constraints needed to represent the problem
- **Average Constraint Size** :- the average size of a constraint, if this is high it indicates that the LP matrix is highly dense and may be problematic.

The goal of building this feature set was to have a small number of highly salient features, due to the problems involved in gathering a large dataset it was essential that the features lacked noise.

5 Machine learning Methods

Initially the author obtained the dataset from previous work. This dataset contained features and run time data for over four thousand problems run on Constraint solvers. The goal of obtaining this data and experimenting with different learning models on it was to gain information about which features from this previous work were strongest. This dataset contained run time data for three constraint solvers; Mistral, Abscon and Choco. Given that Mistral is one of the solvers used in this work the dataset was a useful place to start looking for information to guide this work.

5.1 Initial results on previous dataset

The first approach taken was to learn a ridge regression model on the data. To do this the data was split up into ten randomised subsets and ten fold cross validation was used. The parameter λ was trained across the ten folds. There was a hold out test test of 300 instances that the different models were evaluated on. The ridge regression model yielded an RMS of 433.16 with a $\lambda = 0.1$. This

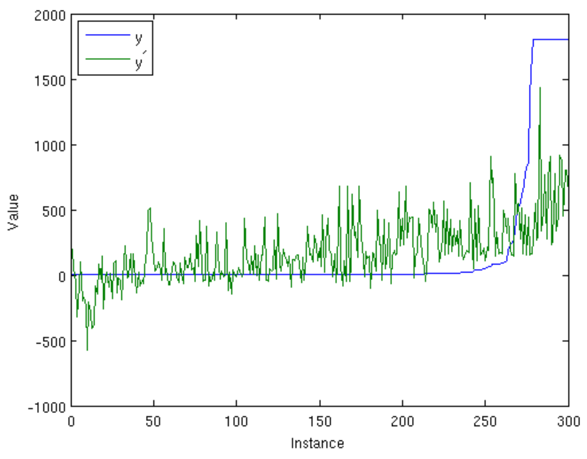

Figure 1: Linear regression model on previous dataset.

small value of lambda indicated that other linear models may be useful. A general linear model with a normal distribution was then fitted, this yielded an RMS of 430.95 on the hold out test set, the performance of this model on the hold out test set can be seen in Figure [1]. This RMS was considered to be very poor. An attempt to remedy this was done by running a wrapper features subset selection [[4]] algorithm on the data. This algorithm is based in using simulated annealing to explore the features subspace. The difference between this algorithm and true simulated annealing is that the wrapper feature subset algorithm terminates early to prevent overfitting. Using this feature selection algorithm a subset of features was selected that yielded an RMS of 371.86. The poor performance of different methods even with intensive feature selection indicated that the dataset was poor. The decision was taken to start from scratch with a new dataset of highly salient features that would be both fast to compute and also give an accurate representation of the difficulty of the problem, as described in Section [4].

5.2 Methods applied to new dataset

Once the new dataset was completed a number of different machine learning methods were applied. Different linear models were fitted with varying results across the different solvers. Many extra features were added by applying non linear functions to the existing features, features selection was computed across these new features. The results of these approaches are detailed in Section [6].

Results predicting runtime

Solver	AverageRMS	GeomRMS	STD error	Baseline
Mistral	161.190987	157.421550	34.179266	164.418495
MiniSat	439.357377	439.318268	6.189424	677.910857
SCIP	382.838617	382.339207	20.597999	445.746684

Table 1: Results for linear regression model predicting absolute runtime.

Results predicting on a log scale

Solver	AverageRMS	GeomRMS	STD error	Baseline
Mistral	1.477267	1.455386	0.248202	1.618452
Minisat	2.955717	2.953969	0.106995	5.253852
SCIP	2.845814	2.843903	0.109730	3.688003

Table 2: Results for linear regression predicting runtime on a log scale.

6 Results

The initial results of running a simple Linear regression on the features gathered to try and predict runtime are shown in Table [6]. These results were obtained by running a ten fold cross validation and averaging the errors in prediction. Three stats are obtained; the arithmetic mean RMS, the geometric mean RMS and the deviation of RMS values. Given the novelty of this work comparing to a baseline presented a challenge. To ensure that the learning methods were working a simple baseline was computed. To obtain this the average of the runtime for the instances in each test set is computed and this value is used for the prediction.

Since runtime is normally measured on a log scale the data was transformed to this scale and the same linear regression test performed. These results are detailed in Table [6]. Again the learning outperforms the baseline across all three solvers.

From this table it is clear to see that the results obtained for runtime on Mistral are far better than the results obtained for the other solvers, despite the other solvers doing comparatively better when compared to the baseline. To attempt to remedy this a number of extra features were added by taking the existing features and applying non linear functions such as x^2, $\sqrt{x}$ and $cos(x)$ and subsequently normalising all features to the range $\{0, 1\}$. No benefit was found to adding in these extra features. To ensure that they were not in fact of benefit to the learning a feature subset selection algorithm was run on the expanded feature set. An algorithm that runs greedily choosing the best feature subset was then applied to the expanded feature set. To ensure that the feature subsets were not over-fitting each subset was tested on a ten fold cross validation and a randomly chosen hold out test set was used as the final judgement. This

Solver	Average RMS
Mistral (New Dataset)	161.19
Mistral (Old Dataset)	229.1

Table 3: RMS reported for linear regression on new and old datasets

feature subset selection was run over a number of number of different random seeds to ensure a fair result when selecting the hold out test set. Again this showed that the additional non linear features add little to the learning.

The final Table [6] shows the results comparing the dataset of [5] ,with the time scaled down to a similar time limit as this work, to the dataset produced by this work. It is clear from this table that the new features produced by this work allow a better prediction of run time than those from previous work.

7 Conclusion

In conclusion this work set out to accurately predict the runtime of solvers on combinatorial problems. The results for predicting the run time of the solver Mistral are promising. The results for predicting the run time of the solvers Mistral and SCIP are still poor in comparison to Mistral and are opportunities for future work.

This work has produced a new dataset of problem features and run time data as well as exploring a novel method of mining feature data from an instance. The results for predicting run time on the solver Mistral have exceeded the run time prediction of an identical model trained on an existing feature set.

References

[1] Emmanuel Hebrard, Eoin O'Mahony, and Barry O'Sullivan. Constraint programming and combinatorial optimisation in numberjack. In Lodi et al. [3], pages 181–185.

[2] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Automated configuration of mixed integer programming solvers. In Lodi et al. [3], pages 186–202.

[3] Andrea Lodi, Michela Milano, and Paolo Toth, editors. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 7th International Conference, CPAIOR 2010, Bologna, Italy, June 14-18, 2010. Proceedings*, volume 6140 of *Lecture Notes in Computer Science*. Springer, 2010.

[4] John Loughrey and Pdraig Cunningham. Overfitting in wrapper-based feature subset selection: The harder you try the worse it gets. In Max Bramer,

Frans Coenen, and Tony Allen, editors, *Research and Development in Intelligent Systems XXI*, pages 33–43. Springer London, 2005.

[5] E. O'Mahony, E. Hebrard, A. Holland, C. Nugent, and B. O'Sullivan. Using case-based reasoning in an algorithm for constraint solving. In *AICS*, 2008.

[6] Olivier Roussel and Christophe Lecoutre. Xml representation of constraint networks: Format xcsp 2.1. *CoRR*, abs/0902.2362, 2009.

[7] N. Tamura, A. Taga, S. Kitagawa, and M. Banbara. Compiling Finite Linear CSP into SAT. pages 590–603, 2006.

[8] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-brown. Satzilla2009: An automatic algorithm portfolio for sat. solver description. In *2009 SAT Competition*, 2009.