Constructing constraint solvers using Monte Carlo Tree Search

Arūnas Prokopas[1] Alan Frisch[2] Ian P. Gent[1] Christopher Jefferson[1]
Lars Kotthoff[3] Ian Miguel[1] Peter Nightingale[1]

[1] University of St Andrews. {ap637,Ian.Gent,caj21,ijm,pwn1}@st-andrews.ac.uk
[2] University of York alan.frisch@york.ac.uk
[3] University College Cork larsko@4c.ucc.ie

Abstract: Constraint solvers are complex pieces of software that are capable of solving a wide variety of problems. Customisation and specialisation opportunities are usually very limited and require specialist knowledge. The Dominion [4] constraint solver synthesizer automatically creates problem-specific solvers. The configuration of a constraint solver is highly complex, especially if the aim is to achieve high performance. We demonstrate how Monte Carlo Tree Search can be employed to tackle this problem.

1 Introduction

Constraints are a natural, powerful means of knowledge representation and reasoning. Dominion is a novel framework for the dynamic and problem-specific generation of constraint solvers. Instead of having a monolithic core with limited configuration options and parameters, Dominion provides the ability to customise every part of a solver. Our approach is considerably more flexible than existing approaches. This additional flexibility comes at a cost though – it is much more difficult to find good solvers.

The main feature that distinguishes Dominion from parameterised solvers is that there is no "default" configuration – Dominion cannot be run at all without configuring a complete solver. This must be done by selecting a large number of components from the component database. During this selection, performance of the components has to be taken into the account, while also keeping track of constraints between components, so that the constructed solver is valid (able to solve the given problem correctly). A complete description of the Dominion architecture can be found in [4]. This configuration is a very difficult problem by itself, therefore standard techniques cannot be applied to it.

There are approaches that attempt similar tasks in the literature, but they almost always rely on background knowledge being available to guide the decision making. Our approach relies on no background knowledge and discovers the performance impact of the various decisions to be made dynamically and for the specific problem to be solved.

2 Monte Carlo Tree Search

Monte Carlo Tree Search [2] (MCTS) is a recent best-first search algorithm that can be applied to wide range of problems that can be expressed as a tree of sequential decisions. The main advantage of MCTS over similar methods is that it can be used with little or no domain knowledge and has shown to be applicable in cases where other algorithms have failed. Because of this, since its appearance, MCTS has been applied to a wide range of complex problems (most notably in various game simulations) [1].

The core of MCTS algorithm is quite simple – every iteration of the algorithm consists of four stages: **selection**, **expansion**, **simulation** and **backpropagation**.

During the **selection** step a *selection strategy* is applied to recursively build the tree from the previously explored nodes balancing between the most promising nodes (*exploitation*) and nodes with a lot of uncertainty (*exploration*). The **expansion** step is used to add new nodes to the partially completed tree at which point a number of **simulations** are run to evaluate the new expansion. The results of those simulations are then **backpropagated** to update the values of the ancestor nodes.

3 Generating Dominion Solvers with Monte Carlo

To synthesize a Dominion solver, we generate its architectural description, one component at a time. The structure of this specification is very hierarchical – the number of choices that have to be made and the choices themselves depend greatly on the previous choices.

During the solver generation two data structures are maintained: a PARTIALTREE, which stores the components we have selected so far, and OPENNODES list, which contains the list of nodes to which we can assign the next component.

During the **selection** stage, we select the most interesting node from the OPENNODES list. This is done by analysing the previous simulations and comparing the runtime averages of solvers featuring each component choice for every node. The node with the largest difference between averages is considered the most interesting.

We then **expand** all of the possible choices at that node by running a number of **simulations** for each of the possible components that can be assigned to that node. The best performing component is assigned to the node and the node itself is moved from the OPENNODES list to the PARTIALTREE. At the same time, its child nodes are added to the OPENNODES list.

Compared to standard MCTS implementations, in the **selection** step we check all components that can be assigned to a single node rather than jumping back and forth between all nodes, which in turn gives a much larger weight

to the *exploitation* aspect of the MCTS over *exploration*. This is preferable behaviour, because arbitrary exploration (combined with complex component constraints) can block the algorithm from choosing important choices that we can detect otherwise. Furthermore, more focus on exploration would result in a much larger number of unnecessary simulations, which are time intensive.

4 Experimental evaluation

Figure 1 shows the progress on a training set of n-Queens problems. Initially, the randomly generated solvers time out for all but the smallest problem instances. However, the information acquired from small problem instances enables us to guide the search towards the configurations with good performance. After a number of iterations, we are able to produce solvers that perform well for all instances.

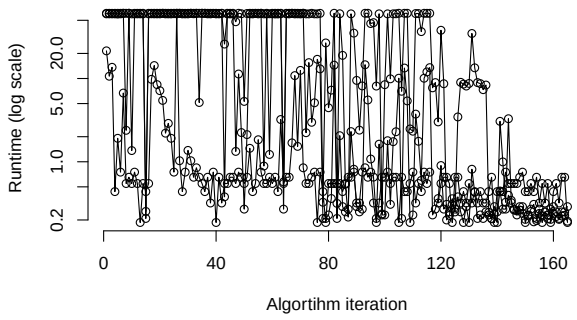

Figure 1: Time it takes to solve 10/50/100-Queens

We compare the performance of the solvers generated by our algorithm to four configurations of Minion [3] constraint solver. It should be noted that both systems can be fine-tuned further, but one of the core features of the dominion is the fact that it should not require any configuration from end user. We chose 4 different heuristics for Minion to demonstrate that this approach still needs to be refined to produce solvers that outperform a hand-tuned traditional solver.

Minion	Worst		Best	
Instsance	Best	Worst	Best	Worst
Magic Squares	**9999.9**	**9999.9**	**9999.9**	**9999.9**
n-Queens	**9999.9**	**1044.5**	**9999.9**	**1044.1**
Disc. Tomography	**100.00**	0.7058	**95.005**	0.1935
Knapsack	**5012.8**	0.0127	**4022.7**	0.0178
Graph Coloring	**251.23**	0.2578	0.7318	0.0428
Sonet	**4.2712**	0.4545	0.4246	0.1010
Golomb Ruler	0.8569	0.2777	0.3476	0.0760

Table 1: Performance vs best/worst Minion configurations

We have used our algorithm to generate a Dominion solver for all of the problems and then compared that solver against Minion (5 to 10 different instances for each problem were tested). For each instance we then selected the best and worst of the four Minion configurations (they vary between problems and instances) performances and compared them to the Dominion instance. The Table 4 shows two instances (best and worst for dominion) and two Minion configurations (best and worst).

As the table demonstrates, this approach already performs exceptionally well for problems that scale very well (*n-Queens* and *Magic Squares*) as it takes advantage of the small instances. If the problem instances are not very predictable (as is the case with *Discrete Tomography* and *Knapsack* problems) and we choose the wrong instances for our training set, the resulting solver can be over-fitted to those instances and it's performance might not be satisfactory when used with unseen instances.

In cases where the solvers are poor in general, we suspect that initial simulations are not providing enough data to make objective choices (the results are either too similar, large number of them time-out) in which case the algorithm attempts to make the best guess (which can naturally be wrong).

5 Conclusions and future work

We have shown the application of Monte Carlo Tree Search to the problem of configuring a Dominion constraint solver. The difficulty of this problem lies not only in the large number of components and their complex interdependencies, but also in the need to configure a solver that exhibits good performance on the problems it was generated for.

As out experiments demonstrate, there remain a number of open challenges. To address the outstanding issues, we are investigating ways of guiding the search to generate diverse sets of solvers instead of random ones. That is, we aim to maximise the configuration differences to facilitate better and more meaningful comparison.

We also are investigating the use of generalized linear regression for rating our nodes (which should give us a better estimate on node importance) and ways to determine when we do not have sufficient information to make a particular (potentially wrong) choice.

References

[1] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1), 2012.

[2] G. M. J. Chaslot. *Monte-Carlo Tree Search*. PhD thesis, Maastricht University, 2010.

[3] Ian P. Gent, Christopher Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. In *Proceedings ECAI 2006*, pages 98–102, 2006.

[4] I. Miguel, D. Balasubramaniam, I. P. Gent, C. Jefferson, T. Kelsey, and S. Linton. A constraint solver syntehsiser: Case for support, 2009.