Constructing constraint solvers using Monte Carlo Tree Search

Arūnas Prokopas\textsuperscript{1}  Alan Frisch\textsuperscript{2}  Ian P. Gent\textsuperscript{1}  Christopher Jefferson\textsuperscript{1}
Lars Kotthoff\textsuperscript{3}  Ian Miguel\textsuperscript{1}  Peter Nightingale\textsuperscript{1}

\textsuperscript{1} University of St Andrews. \{ap637,Ian.Gent,caj21,ijm,pwn1\}@st-andrews.ac.uk
\textsuperscript{2} University of York alan.frisch@york.ac.uk
\textsuperscript{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\textsuperscript{[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 offer a natural, powerful means of representing and reasoning about combinatorial problems. For example, in the production of a bus driver schedule several constraints occur, such as: each bus must have exactly one driver; a driver cannot be assigned to more than one bus at once; every driver must have the requisite number of breaks in his/her shift. Constraint solving offers a means by which solutions to such problems can be found automatically. Modern constraint solvers are sophisticated pieces of software that require significant manual tuning to solve large, complex problems successfully.

The Dominion constraint solver synthesises a bespoke solver from scratch for that problem from a library of components. This allows the automatic customisation of every aspect of the solver to suit the input problem.

The key challenge for this approach is how to select among the many solvers Dominion can synthesise for a given problem. A large number of components must be selected from the component library. In making this selection, the performance of the components must 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).

The approach we describe in this paper 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\textsuperscript{[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)\textsuperscript{[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 synthesise a Dominion solver, we generate its architectural description, one component at a time (memory managers, variable representations, constraint propagators etc.). The structure of this specification is very hierarchical – the number and types of choices that have to be made depend greatly on the previous choices.

During the solver generation two data structures are maintained: a PARTIALTREE, which stores the nodes with 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](image)

We compare the performance of the solvers generated by our algorithm to four configurations of the Minion [3] constraint solver (to be specific, we used four different variable orderings - static, SDF, WDEG, dom/WDEG). It should be noted that both systems can be fine-tuned further, but one of the core features of Dominion is the fact that it should not require any configuration from end user. Nevertheless, 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.

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

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 its performance might not be satisfactory when used with unseen instances.

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

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.

While MCTS algorithm can potentially generate very good solvers, as our experiments demonstrate that the current implementation relies on chance too much and still lacks consistency. To address this, we are currently investigating how could we generate more diverse sets of tests (as opposed to random generation) and how could we evaluate and improve the confidence of the initial choices.

### References