# Constructing problem-specific constraint solvers using Monte Carlo Tree Search

Arūnas Prokopas, Ian Gent, Ian Miguel

University of St Andrews

**Abstract** Constraint solvers are complex pieces of software that often offer a few customisation opportunities and require specialist knowledge in order to optimise them. The Dominion constraint solver synthesizer addresses these issues by automatically building problem-specific solvers. This paper demonstrates that Monte Carlo Tree Search can be employed to tackle the difficult problem of configuring these solvers.

## 1 Introduction

Since constraint solvers must be able to handle a wide range of unseen problems, modern constraint solvers are sophisticated and complex pieces of software. This means that in order to optimise the solver for a particular large, complex problem, a significant amount of manual tuning is often required.

The Dominion constraint solver synthesiser [1] provides an alternative approach. Rather than providing a fixed core solver with limited configuration options, it analyses the input problem and synthesises a solver for that particular problem from the library of available components. This approach allows the automatic customisation of every aspect of the solver to suit the input problem.

The main problem with this approach is that a large number of components have to be selected for each solver and the number of different configurations increases exponentially with the problem complexity and the library size. Additionally, these components may have an arbitrary number of subcomponents and internal constraints to ensure the validity of solvers (ability to solve the given problems correctly).

Since the shape and size of the component (and in turn decision) tree can vary significantly for every problem, it is difficult to apply the standard tuning methods to it and a specialised algorithm is needed. In this paper we demonstrate how Monte Carlo Tree Search[6] (MCTS) can be used for this purpose.

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 Dominion

Dominion consists of a database of components, which are described in the architecture description language (ADL) Grasp [3]. A full description of how Dominion uses Grasp to specify solvers can be found in [2].

Grasp is designed to capture the structure, behaviour and rationale of systems at the architectural level and supports a range of architectural primitives. This is needed to represent the complex relationships between different components in Dominion. Each component in Dominion is represented by a Grasp template, which expresses the interfaces the component provides and the child components which it requires. Further, Grasp allows a range of constraints to be expressed between components. A complete solver forms a tree of Grasp templates.

```
@Dominion(<external annotation>)
template <identifier>(<variables>){
    provides <function>;
    requires <function> <id>;
    property <id> = <value>;
    check <list> subsetof <list>;
    link <component> to <variable>;

}
```

Figure 1: Grasp Template

```
@Dominion(Filename='nqueens.hpp')
template DominionProblem() {
    provides IProblemClass;
    requires IPropagator_sum sum1;
    requires IDiscreVar queens;
    . . .
    link queens to sum1.var1;
    check ['bound'] subsetof
                queens.domainType;
}
```

Figure 2: Grasp Component Example

In general, different templates which implement the same interface can have very different implementations. For example, Dominion contains five implementations of the $\sum x_i = d$ constraint. Some of these impose extra requirements on the variables, and different implementations perform better or worse as the number of variables increases.

An initial algorithm analyses the input problem and determines what kind of algorithms are needed to solve the given problem, what storage is needed for the variables within the problem and how they are all linked together. These requirements are then noted down as a Grasp specification to be used in the solver synthesis.

The solver synthesis is a configuration problem [7] by itself . At the root of this configuration tree is always the same component (ISolver) that provides the main interface and requires the problem specification (produced during the analysis step), a search algorithm and a search heuristic. Making a choice for each of these components opens up the further component choices. As different components, which implement the same interface, can have different requirements and impose different constraints, the shape of the resulting component tree can vary very significantly, even when constructing solvers for a very specific constraint problem. This means that it is often not feasible to list all the decisions, which have to be made during the configuration, in advance and in turn it makes the application of standard tuning techniques very difficult.

## 3 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a recent best-first search algorithm that can be applied to a 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 [5]. Because of this, since its appearance, MCTS has been applied to a wide range of complex problems (most notably in various game simulations[4,8,10]).

The concepts of the MCTS algorithm are 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.

## 4 Implementation

Given that the completed solver is a tree of components of undefined shape that is filled as the search progresses, our Monte Carlo Tree Search algorithm relies on the following data-structures:

- **PartialTree**: This is the list of components we have assigned so far. Initially this contains only the ISolver component, which is the root of all Dominion solvers.
- **OpenNodes**: The list of Grasp interfaces to which we can assign the next component. Initially this contains the three interfaces that are required by the ISolver
- **PreviousTests**: The specifications and runtimes of all solvers that are tested so far. This list starts empty.

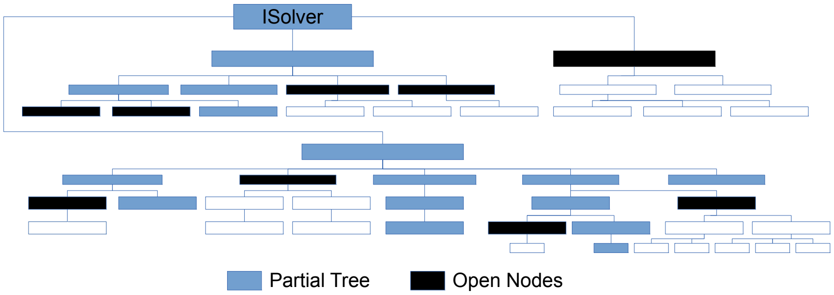

Figure 3: Dominion Solver Component Tree

At the core of the MCTS algorithm is the **Evaluation** subroutine. It generates and tests a number of random solvers by extending the **PartialTree** (which can be treated asi a MCTS node). During this generation, the Grasp specification is evaluated, components are filtered by their function and assigned randomly. If any of the constraints become violated after the assignment, the algorithm backtracks. All the successful tests and their runtimes are then appended to the **PreviousTests** list. This can be repeated with multiple parameter sets to avoid overfitting.

### 4.1 Core Algorithm

The main algorithm begins with the **Evaluation** subroutine, to generate some initial data to guide the rest of the algorithm. After which we enter the main loop of the algorithm.

Each algorithm iteration begins with the **Interest** function. This function iterates over each element in the **OpenNodes** list and checks what possible assignments can be made for each of those nodes. If the element can only have one possible assignment, that component is immediately noted in the **PartialTree** and its child nodes are added to the **OpenNodes** list.

If there are multiple possible component assignments for that particular Node, **PreviousTests** list is examined to find all tests, that fit **PartialTree** to see which assignments were previously made. The difference between the best and worst assignment is noted as the **Interestingness** of the node.

After the **Interest** function finishes evaluating the nodes within the **OpenNodes** list, the node with the highest **Interestingness** is selected. For each of the possible assignments to that particular node, the assignment is made temporarily and the **Evaluation** subroutine is run. This provides additional information, which is used to select the best component.

At this point, the best component is chosen and the assignment is made within the **PartialTree**, the child components of that node are added to the **OpenNodes** list and the algorithm moves to the next iteration.

### 4.2 Restarts

One very significant flaw inherent to this approach is the fact that it favours exploitation very heavily and does very little in terms of exploration. As a result of this, if not enough data is gathered by the initial runs of the **Evaluation** subroutine, it can make some terrible decisions and never reach the desired results.

Refalo [9] demonstrates, that in cases like this, where an algorithm relies heavily on random selection, restarting the search can dramatically improve the performance of the algorithm. To implement this, the **Evaluation** algorithm was extended to check the number of tests done so far and initiate a **Restart** when the predefined threshold is reached.

The **Restart** subroutine resets the **PartialTree** and **OpenNodes** list, but maintains the **PreviousTests** list. It then starts the whole MCTS algorithm. Since it very likely that the new iteration would follow the same path (because of all the good solvers already discovered there), the initial **Evaluation** algorithm is run again as well.

## 5 Evaluation and Problems Encountered

One of the biggest problems inherent to this approach is the fact that a large number of solvers have to be compiled and tested. Additionally, running each of these solvers can be a very expensive operation. These two factors, when combined, mean that in may take a prohibitively long time to find a good solver for any particular problem class.

This is especially true for the more complex problems, which do not have relatively easy instances.

If the problems have both easy and hard instances, and collection of these can be assembled, a large number of tests with very small timeouts can be used to sample the components. Additionally, if the instances are already sorted by difficulty, the testing can be further sped up by assuming time-outs for the remaining instance after the first time-out is encountered.

In cases where it is possible to apply the both techniques it is already possible to achieve very good results as shown in the Table 1. In case of N Queens, the Dominion was able to build solvers that can scale extremely well up to 470 queens, while Minion with standard configurations was struggling above 30.

| Problem Class | Dominion | Minion1 | Minion2 | Minion3 | Minion4 |
|---:|---:|---:|---:|---:|---:|
| 20 Queens | 0.017 | 0.721 | 0.128 | 0.120 | 0.112 |
| 30 Queens | 0.032 | 105.1 | 0.447 | 109.6 | 0.578 |
| 40 Queens | 0.066 | 599.1 | 599.6 | 599.7 | 599.6 |
| Magic Squares 5 | 0.102 | 0.337 | 0.115 | 0.428 | 0.123 |
| Magic Squares 6 | 0.207 | 599.7 | 26.84 | 133.8 | 599.7 |
| Magic Squares 7 | 1.817 | 599.6 | 599.7 | 599.7 | 599.7 |

Table 1: Runtimes: Dominion vs Minion with different variable orderings
Minion search orders: standard, SDF, WDEG, domWDEG

Another problem emerges from the random sampling of the solver space. While the algorithm is guaranteed to reach the good solvers eventually, “unlucky” samples can delay this process tremendously. Figure 4 demonstrates that even for simple problems like N Queens, a few “lucky” random configurations is not enough to identify the component choices that are needed to reach the good solver configurations. (The vertical bars in the graphs represent restarts and subsequent random sampling.)

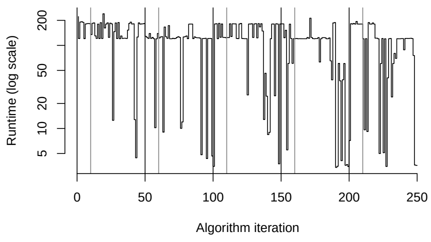

Figure 4: 10/20/100/200 Queens, 60s t/o

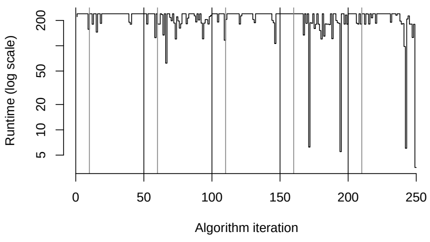

Figure 5: Magic Squares (5/7/9/11), 60s t/o

This generally happens for two reasons:

- Some components are not present in any of the tests. This leads to the **Interest** function ignoring the important components.
- Some component combinations occur together too often. This leads to the **Interest** function and subsequent checks choosing the wrong components.

To address this issue, we are currently investigating the possibility of modifying the **Evaluation** subroutine so it would no longer generate solvers randomly. Instead, the process will be guided to eliminate the missed components and increase the number of different component combinations. Additionally, the naive **Interest** function that checks best tests for each of the components will be replaced by more robust techniques like Generalised Linear Regression models.

## 6 Conclusion

While there is still more work to be done to increase the scalability of the approach, the early results demonstrate that this approach can be used to effectively (if not yet efficiently) build specialised solvers that can outperform the standard monolithic solvers.

**Acknowledgements** We would like to thank EPSRC for funding this work through grant EP/H004092/1. We thank Christopher Jefferson and Peter Nightingale for constructing the large portion of the underlying Dominion framework and Alan Frisch for suggesting the Monte Carlo Tree Search simulation.

## References

1. Balasubramaniam, D., Jefferson, C., Kotthoff, L., Miguel, I., Nightingale, P.: An automated approach to generatin efficient constraint solvers. In: Proceedings ICSE 2011 (2011)
2. Balasubramaniam, D., Jefferson, C., Kotthoff, L., Miguel, I., Nightingale, P.: An automated approach to generating efficient constraint solvers. In: 34th International Conference on Software Engineering. pp. 661–671 (Jun 2012)
3. Balasubramaniam, D., de Silva, L.: Grasp language reference manual version 1.0. Tech. rep., University of St Andrews (2011), http://www.cs.st-andrews.ac.uk/~dharini/reports/GraspManual.pdf
4. Broeck, G.V., Driessens, K., Ramon, J.: Monte-carlo tree search in poker using expected reward distributions pp. 367–381 (2009)
5. Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of monte carlo tree search methods. IEEE Transactions on Computational Intelligence and AI in Games 4(1) (2012)
6. Chaslot, G.M.J.: Monte-Carlo Tree Search. Ph.D. thesis, Maastricht University (2010)
7. Günter, A., Kühn, C.: Knowledge-based configuration- survey and future directions pp. 47–66 (1999)
8. I. Szita an, G.C., Spronck, P.: Monte-carlo tree search in settlers of catan pp. 21–32 (2010)
9. Refalo, P.: Impact-based search strategies for constraint programming. In: CP 2004. pp. 557–571 (2004)
10. Winands, M.H.M., Björnsson, Y., Saito, J.T.: Monte-carlo tree search solver pp. 25–36 (2008)