An empirical comparison of the structure of configuration files of continuous integration and build systems

Joseph Ling

jl653@kent.ac.uk

School of Computing
University of Kent
United Kingdom

Word Count: 6,100

January 28, 2020

Abstract

This paper describes a simple heuristic approach to solving large-scale constraint satisfaction and scheduling problems. In this approach one starts with an inconsistent assignment for a set of variables and searches through the space of possible repairs. The search can be guided by a value-ordering heuristic, the *min-conflicts heuristic*, that attempts to minimize the number of constraint violations after each step. The heuristic can be used with a variety of different search strategies. We demonstrate empirically that on the n-queens problem, a technique based on this approach performs orders of magnitude better than traditional backtracking techniques. We also describe a scheduling application where the approach has been used successfully. A theoretical analysis is presented both to explain why this method works well on certain types of problems and to predict when it is likely to be most effective.

1 Introduction

need more introduction into CI like the travis paper probably?

For continuous integration (CI) and build systems/continuous delivery a lot of it is configured with code as configuration. The main kind of configuration format used for this is yaml (reference to what it is??) followed by xml and java based scripting formats. As CI has become increasingly more popular and practical ()................For software development teams there are roles for just looking after and managing CI and the deployment infrastructure. Understanding how code as configuration is used for CI is an emerging area of research because of this.

cite needed here to backup

The first stage in order to answer this question is to get a dataset of configuration files from different ci/cd systems. In doing so answering the questions of: - what is the spread of ci/cd systems for public github repositories? this will take into account operating system, programming language, star count, subscriber count - note: something along the lines of multiple configuration files - what naming convention do they use for the files? (in order to understand common practices)

After getting all that data then it will be need to analysed to how it is used. The key questions for this that will be focused on will be: - how are comments used in the configuration? - how are stages (named sections of the process) named for configuration files? - branch names - configuration errors when loading the config (just yaml parsing errors atm) - how scripts with the configuration files? (need to elloborate more on this one)

A key aspect is that these questions do not look too deeply into the individual implementation of each of unique piece of configuration for each type. This is because there are already some good papers looking Gallaba and McIntosh (2018) at this but in order to be able to compare the different configuration types it is important to compare similar attributes (there is also a time factor in here as well).

need to carefully plan out how to gage the size of repo, as I don't want to have too download repos

2 Previous Works

Config as code is not necessarily infrastructure as code is slightly differently. So the comparison isn't fair necessarily or accurate.

Configuration as code or infrastructure as code has been an increasing area of research over the last few years. There seems to be slightly more research in infrastructure as code Rahman, Mahdavi-Hezaveh and Williams (2019). The has been a focus on Puppet and Chef, for example in Sharma, Fragkoulis and Spinellis (2016) looks at code quality by the measure of "code smell" of Puppet code. This tackles the problem by defining by best practises and analysing the code against that. In the case of Cito et al. (2017) it uses the docker linter in order to be able to analyse the files.

15, 18, 23, 30

docker, puppet, travis analysis

3 Methodology

foo bar asdffsdf

3.1 first load

By almost any measure, the Hubble Space Telescope scheduling problem Between ten thousand and thirty thousand astronomical observations per year must be scheduled, subject to a great variety of constraints including power restrictions, observation priorities, time-dependent orbital characteristics, movement of astronomical bodies, stray light sources, etc. Because the telescope is an extremely valuable resource with a limited lifetime, efficient scheduling is a critical concern. An initial scheduling system, developed using traditional programming methods, highlighted the difficulty of the problem; it was estimated that it would take over three weeks for the system to schedule one week of observations. As described in section 5, this problem was remedied by the development of a successful constraint-based system to augment the initial system. At the heart of the constraint-based system is a neural network developed by Adorf and Johnston, the Guarded Discrete Stochastic (GDS) network,

From a computational point of view the network is interesting because

Adorf and Johnston found that it performs well on a variety of tasks, in addition to the space telescope scheduling problem. For example, the network performs significantly better on the n-queens problem than methods that were previously developed. The n-queens problem requires placing n queens on an $n \times n$ chessboard so that no two queens share a row, column or diagonal. The network has been used to solve problems of up to 1024 queens, whereas most heuristic backtracking methods encounter difficulties with problems one-tenth

In a standard Hopfield network, all connections between neurons are symmetric. In the GDS network, the main network is coupled asymmetrically to an auxiliary network of *guard neurons* which restricts the configurations that the network can assume. This modification enables the network to rapidly find a solution for many problems, even when the network is simulated on a serial machine. Unfortunately, convergence to a stable configuration is no longer guaranteed. Thus the network can fall into a local minimum involving a group of unstable states among which it will oscillate. In practice, however, if the network fails to converge after some number of neuron state transitions, it can simply be stopped and started over.

To illustrate the network architecture and updating scheme, let us consider how the network is used to solve binary constraint satisfaction problems. A problem consists of n variables, $X_1 \ldots X_n$, with domains $D_1 \ldots D_n$, and a set of binary constraints. Each constraint $C_\alpha(X_j, X_k)$ is a subset of $D_j \times D_k$ specifying incompatible values for a pair of variables. The goal is to find an assignment for each of the variables which satisfies the constraints. (In this paper we only consider the task of finding a single solution, rather than that of finding all solutions.) To solve a CSP using the network, each variable is represented by a separate set of neurons, one neuron for each of the variable's possible values. Each neuron is either "on" or "off", and in a solution state, every variable will have exactly one of its corresponding neurons "on", representing the value of that variable. Constraints are represented by inhibitory (i.e., negatively weighted) connections between the neurons. To insure that every variable is assigned a value, there is a guard neuron for each set of neurons representing a variable; if no neuron in the set is on, the guard neuron will provide an excitatory input that is large enough to turn one on. (Because of the way the connection weights are set up, it is unlikely that the guard neuron will turn on more than one neuron.) The network is updated on each cycle by randomly picking a set of neurons that represents a variable, and flipping the state of the neuron in that set whose input is *most inconsistent* with its current output (if any). When all neurons' states are consistent with their input, a solution is achieved.

To solve the n-queens problem, for example, each of the $n \times n$ board positions is represented by a neuron whose output is either one or zero depending on whether a queen is currently placed in that position or not. (Note that this is a local representation rather than a distributed representation of the board.) If two board positions are inconsistent, then an inhibiting connection exists between the corresponding two neurons. For example, all the neurons in a column will inhibit each other, representing the constraint that two queens cannot be in the same column. For each row, there is a guard neuron connected to each of the neurons in that row which gives the neurons in the row a large excitatory input, enough so that at least one neuron in the row will turn on. The guard neurons thus enforce the constraint that one queen in each row must be on. As described above, the network is updated on each cycle by randomly picking a row and flipping the state of the neuron in that row whose input is most inconsistent with its current output. A solution is realized when the output of every neuron is consistent with its input.

4 Why does the GDS Network Perform So Well?

Our analysis of the GDS network was motivated by the following question: "Why does the network perform so much better than traditional backtracking methods on certain tasks"? In particular, we were intrigued by the results on the n-queens problem, since this problem has received considerable attention from previous researchers. For n-queens, Adorf and Johnston found empirically that the network requires a linear number of transitions to converge. Since each transition requires linear time, the expected (empirical) time for the network to find a solution is $O(n^2)$. To check this behavior, Johnston and Adorf ran experiments with n as high as 1024, at which point memory limitations became a problem.[1]

4.1 Nonsystematic Search Hypothesis

Initially, we hypothesized that the network's advantage came from the nonsystematic nature of its search, as compared to the systematic organization inherent in depth-first backtracking. There are two potential problems associated with systematic depth-first search. First, the search space may be organized in such a way that poorer choices are explored first at each branch point. For instance, in the n-queens problem, depth-first search tends to find a solution more quickly when the first queen is placed in the center of the first row rather than in the corner; apparently this occurs because there are more solutions with the Nevertheless, most naive algorithms tend to start in the corner simply because humans find it more natural to program that way. However, this fact by itself does not explain why nonsystematic search would work so well for n-queens. A backtracking program that randomly orders rows (and columns within rows) performs much better than the naive method, but still performs poorly relative to the GDS network.

The second potential problem with depth-first search is more significant and more subtle. As illustrated by figure 1, a depth-first search can be a disadvantage when solutions are not evenly distributed throughout the search space. In the tree at the left of the figure, the solutions are clustered

[1] The network, which is programmed in Lisp, requires approximately 11 minutes to solve the 1024 queens problem on a TI Explorer II. For larger problems, memory becomes a limiting factor because the network requires approximately $O(n^2)$ space. (Although the number of connections is actually $O(n^3)$, some connections are computed dynamically rather than stored).

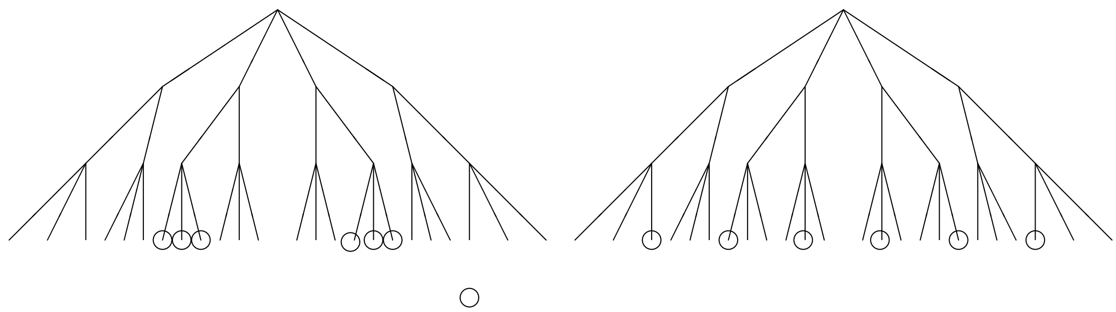

Figure 1: Solutions Clustered vs. Solutions Evenly Distributed

together. In the tree on the right, the solutions are more evenly distributed. Thus, the average distance between solutions is greater in the left tree. In a depth-first search, the average time to find the first solution increases with the average distance between solutions. Consequently depth-first search performs relatively poorly in a tree where In comparison, a search strategy which examines the leaves of the tree in random order is unaffected by solution clustering.

We investigated whether this phenomenon explained the relatively poor performance of depth-first search on n-queens by experimenting with a randomized algorithm begins by selecting a path from the root to a leaf. To select a path, the algorithm starts at the root node and chooses one of its children with equal probability. This process continues recursively until a leaf is encountered. If the leaf is a solution the algorithm terminates, if not, it starts over again at the root and selects a path. The same path may be examined more than once, since no memory is maintained between successive trials.

The Las Vegas algorithm does, in fact, perform better than simple depth-first search on n-queens However, the performance of the Las Vegas algorithm is still not nearly as good as that of the GDS network, and so we concluded that the systematicity hypothesis alone cannot explain the network's behavior.

4.2 Informedness Hypothesis

Our second hypothesis was that the network's search process uses information about the current assignment that is not available to a constructive backtracking program. 's use of an iterative improvement strategy guides the search in a way that is not possible with a standard backtracking algorithm. We now believe this hypothesis is correct, in that it explains why the network works so well. In particular, the key to the network's performance appears to be that state transitions are made so as to reduce the number of outstanding inconsistencies in the network; specifically, each state transition involves flipping the neuron whose output is most inconsistent with its current input. From a constraint satisfaction perspective, it is as if the network reassigns a value for a variable by choosing the value that violates the fewest constraints. This idea is captured by the following heuristic:

> **Min-Conflicts heuristic:**
> *Given:* A set of variables, a set of binary constraints, and an assignment specifying a value for each variable. Two variables *conflict* if their values violate a constraint.
> *Procedure:* Select a variable that is in conflict, and assign it a value that minimizes the number of conflicts. (Break ties randomly.)

We have found that the network's behavior can be approximated by a symbolic system that uses the min-conflicts heuristic for hill climbing. The hill-climbing system starts with an initial assignment generated in a preprocessing phase. At each choice point, the heuristic chooses a variable that is currently in conflict and reassigns its value, until a solution is found. The system thus searches the space of possible assignments, favoring assignments with fewer total conflicts. Of course, the hill-climbing system can become "stuck" in a local maximum, in the same way that the network may become "stuck" in a local minimum. In the next section we present empirical evidence to support our claim that the min-conflicts approach can account for the network's effectiveness.

```
Procedure INFORMED-BACKTRACK (VARS-LEFT VARS-DONE)
 If all variables are consistent, then solution found, STOP.
 Let VAR = a variable in VARS-LEFT that is in conflict.
 Remove VAR from VARS-LEFT.
 Push VAR onto VARS-DONE.
 Let VALUES = list of possible values for VAR in ascending order according
             to number of conflicts with variables in VARS-LEFT.
 For each VALUE in VALUES, until solution found:
   If VALUE does not conflict with any variable that is in VARS-DONE,
   then Assign VALUE to VAR.
        Call INFORMED-BACKTRACK(VARS-LEFT VARS-DONE)
   end if
 end for
end procedure

Begin program
 Let VARS-LEFT = list of all variables, each assigned an initial value.
 Let VARS-DONE = nil
 Call INFORMED-BACKTRACK(VARS-LEFT VARS-DONE)
End program
```

Figure 2: Informed Backtracking Using the Min-Conflicts Heuristic

There are two aspects of the min-conflicts hill-climbing method that distinguish it from standard CSP algorithms. First, instead of incrementally constructing a consistent partial assignment, the min-conflicts method *repairs* a complete but inconsistent assignment by reducing inconsistencies. Thus, it uses information about the current assignment to guide its search that is not available to a standard backtracking algorithm. Second, the use of a hill-climbing strategy rather than a backtracking strategy produces a different style of search.

4.2.1 Repair-Based Search Strategies

(This is a example of a third level section.) Extracting the method from the network enables us to tease apart and experiment with its different components. In particular, the idea of repairing an inconsistent assignment can be used with a variety of different search strategies in addition to hill climbing. For example, we can backtrack through the space of possible repairs, rather than using a hill-climbing strategy, as follows. Given an initial assignment generated in a preprocessing phase, we can employ the min-conflicts heuristic to order the choice of variables and values to consider, as described in figure 2. Initially, the variables are all on a list of VARS-LEFT, and as they are repaired, they are pushed onto a list of VARS-DONE. The algorithm attempts to find a sequence of repairs, such that no variable is repaired more than once. If there is no way to repair a variable in VARS-LEFT without violating a previously repaired variable (a variable in VARS-DONE), the algorithm backtracks.

Notice that this algorithm is simply a standard backtracking algorithm augmented with the min-conflicts heuristic to order its choice of which variable and value to attend to. This illustrates an important point. The backtracking repair algorithm incrementally extends a consistent partial assignment (i.e., VARS-DONE), as does a constructive backtracking program, but in addition, uses information from the initial assignment (i.e., VARS-LEFT) to bias its search. Thus, it is a type of *informed backtracking*. We still characterize it as repair-based method since its search is guided by a complete, inconsistent assignment.

5 Experimental Results

[section ommitted]

6 A Theoretical Model

[section ommitted]

7 Discussion

[section ommitted]

8 Acknowledgement

The authors wish to thank Hans-Martin Adorf, Don Rosenthal, Richard Franier, Peter Cheeseman and Monte Zweben for their assistance and advice. We also thank Ron Musick and our anonymous reviewers for their comments. The Space Telescope Science Institute is operated by the Association of Universities for Research in Astronomy for NASA.

Appendix A. Probability Distributions for N-Queens

[section ommitted]

References

Cito, J., Schermann, G., Wittern, J. E., Leitner, P., Zumberi, S. and Gall, H. C. (2017). An Empirical Analysis of the Docker Container Ecosystem on GitHub. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 323–333, iSSN: null.

Gallaba, K. and McIntosh, S. (2018). Use and Misuse of Continuous Integration Features: An Empirical Study of Projects that (mis)use Travis CI. *IEEE Transactions on Software Engineering*, pp. 1–1.

Rahman, A., Mahdavi-Hezaveh, R. and Williams, L. (2019). A systematic mapping study of infrastructure as code research. *Information and Software Technology*, 108, pp. 65–77.

Sharma, T., Fragkoulis, M. and Spinellis, D. (2016). Does Your Configuration Code Smell? In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pp. 189–200, iSSN: null.