Rocketbear: A Forward Checking Constraint Solver for Binary and Unary Constraints Written in Python

120013913

April 14, 2015

1 Introduction

In this practical we are asked to develop a forward checking constraint solver that uses d-way branching in a language of our choosing. We are also asked to decide on a representation for the constraint and a mechanism for undoing pruning upon backtracking. We were also tasked with implementing various heuristics for variable orderings and to provide empirical evidence on their merit. In order to measure the performance of the solver and the heuristics, we are asked to choose a program class to perform benchmark tests on.

I have used the Python programming language to implement this constraint solver dubbed Rocketbear. I have provided support for unary constraints on variables and binary constraints between variables. Three dynamic and two static heuristics have been implemented. The dynamic heuristics include highest degree first, minimum domain cardinality first, and minimum ratio of domain cardinality to degree. Static analogues of these heuristics are provided for the former two. I have chosen to use graph colouring as the problem instance to benchmark the solver and have used a variety of sample graphs found online with the implemented heuristics to provide evidence of the solver's performance. This report is structured by first describing the representation of variables by Rocketbear, then the representation of constraints, the heuristics, transforming graph colouring into a Rocketbear constraint graph, the results, and a discussion.

2 Variables

Rocketbear represents variables using one's full domain, pruned domain, name, assignment, and unary constraints. The full domain is all the possible values that the variable can be assigned. The pruned domain is dynamic and changes as forward checking progresses. The name of the variable can be anonymous and automatically assigned by the library or can be explicitly stated. The assignment is also a dynamic feature that is used by the forward checking solver to update the variables value. The unary constraints on a variable are added once the variable has been instantiated.

3 Constraints

Both binary and unary constraints are supported by Rocketbear. Unary constraints are added directly to each variable. A unary constraint can be seen as a function that takes in a variable assignment and returns true if the constraint is satisfied. A variable may have more than one unary constraint and these are added into a list of unary constraint maintained by each variable. Binary constraints are represented by functions which take in two parameters, the left and right assignments of the variables on an arc, and returns true if the constraint is satisfied. An arc between variables i and j is added to the constraint graph if there is at least one constraint between i and j. The binary constraint between i and j is stored in the arc object between i and j.

Besides simple binary and unary constraints, the macro-constraint, *AllDifferent* was added. The *AllDifferent* constraint synthesizer took an arbitrary number of variables in as parameters, and added a not equals binary constraint from each variable to every other variable in the constraint graph. Even though the *AllDifferent* is not used in the graph colouring transformation described in Sec. 5, an example is shown in `examples/solver_example.py`.

Also, minimizing capabilities are provided by Rocketbear by specifying a number k such that all variables must be less than k. This is done by adding unary constraints on all variables indicating that their value must be less than a given value k. It is also possible to use unary constraints to specify minimization properties for subsets of variables.

4 Heuristics

Five heuristics have been implemented on the variable ordering. Three are dynamic and two are static. A static heuristic provides a single unchanging ordering for the variables that are expanded in forward checking. A dynamic ordering is dependent on features such as the size of the pruned domains, or the number of valid edges in the constraint graph left for a variable. This dynamic orderings change as the search progresses. The five heuristics are described below.

4.1 Static Most Arcs First

This heuristic will order the variables based on the number of arcs in the constraint graph a certain variable has. Specifically, this heuristic expands variables with the most number arcs (i.e. the most number of binary constraints) first in order to reduce the size of the search space.

4.2 Static Smallest Domain First

This heuristic will order the variables based on the cardinality of their initial domains. Specifically the variable that has the largest initial domain will be expanded first when expanding the search tree.

4.3 Dynamic Most Arcs First

This heuristic is a variant of its static counter part, except with on difference. Instead of counting the total number arcs leading to a variable and letting that determine its ordering, only the number of arcs that have not yet been assigned count towards the ordering. Therefore if a node has five arcs associated with it but three have already been assigned, the evaluation of the heuristic will be two since this heuristic does not take into account neighbouring variables that have already been assigned.

4.4 Dynamic Smallest Domain First

This heuristic is similar to the static smallest domain first, except instead of counting the full domain of a variable and letting that determine its heuristic evaluation, the pruned domain is used. This means that a variable that originally had a domain cardinality of ten but has been pruned and revised down to three, the heuristic evaluation of the variable will be three.

4.5 Dynamic Smallest Domain Over Degree First

This heuristic is a combination of the static most arcs first and the dynamic smallest domain first heuristics. This heuristic evaluates the cardinality of the pruned domain and divides it by the amount of arcs in total a variable has. Since static most arcs first heuristic gives higher heuristic values for variables with more arcs and the dynamic smallest domain first gives higher heuristic evaluations for smaller cardinality domains, by dividing these two heuristics, specifically the cardinality of the pruned domain by the number of arcs a variable has, this heuristic is a good combination of two other indicators for variable ordering.

5 Graph Colouring

The instance problem I have chosen to experiment on is graph colouring. In order to perform graph colouring using Rocketbear, the graph to be coloured is converted in to a constraint graph. This is done by creating an empty constraint graph, iterating over the edges for the graph to be coloured, and adding a constraint indicating that the two nodes on the edge of the graph cannot be equal to the constraint graph. The domains of the variables in the newly generated constraint graph are all equal in cardinality and value and are comprised of the integers from 0 to k where k is the maximum amount of colours to be used in the graph colouring and is given by the user when the colouring method is called. If the graph cannot be coloured using k colours, the method returns a null pointer.

6 Results

For the experiments, three different graphs representing the n-queens problem are used. Each of the five heuristics for variable ordering are tested on each graph. The number of nodes expanded, the number of times backtracking is needed, the number of assignments, and the amount of time it took to find a solution are gathered for each heuristic on each graph. Each one of the experiment instances ran as a separate process on the school server, `lyrane.cs.st-andrews.ac.uk`. This data from the different graphs using varying heuristics is shown in the tables below. The acronyms are used to distinguish between heuristics. If a heuristic does not have a value in the table, it exceeded the time-out of one hour. For each of the tables, the corresponding graph is located in the `graphs/` directory.

From the tables it is apparent that the Dynamic Smallest Domain First heuristic expands the least amount of nodes, uses the least amount of assignments, and backtracks the least out of all of the graph instances that were tested. Likewise, the for all but one graph instance, it produced a solution in the last amount of time. For the Queen5_5 graph, the Static Most Arcs First heuristic produced a solution in the smallest amount of time, however for the other two graph instances, it either did not produce a solution within an hour or took the longest to solve. Likewise for the first graph, the Dynamic Most Arcs First heuristic took by far the longest amount of time to produce solution and it expanded the most nodes, had the most assignments, and the most backtracks. For the second graph instance, the Static Smallest Arcs First had the most assignments, nodes, and backtracks and took the longest to compute. For the last graph instance, none of the heuristics using solely the arcs were able to produce a solution in the provided time frame.

From these tables, it is apparent that the Dynamic Smallest Domain First heuristic produces solutions in the least amount of time, by having the least amount of assignments, nodes expanded, and backtrackings. The Static Smallest Domain First and Dynamic Domain Over Degree heuristics produced the same assignments, nodes, and number of times backtracking was needed for each of the graph instances. They also took roughly the same amount of time for each instance.

Heuristic	Time [sec]	Assignments	Nodes	Backtracks
smaf	2.16678380966	30	27	5
dsdf	2.47054815292	25	25	0
ssdf	3.96269488335	28	26	3
ddod	4.37844085693	28	26	3
dmaf	79.5878660679	1437	742	1412

Figure 1: Table showing data for the Queen5_5 graph

Heuristic	Time [sec]	Assignments	Nodes	Backtracks
dsdf	111.44510603	1459	1282	1423
ddod	420.716995001	7753	5417	7717
ssdf	424.589070082	7753	5417	7717
smaf	683.094563007	10699	7298	10663

Figure 2: Table showing data for the Queen6_6 graph

Heuristic	Time [sec]	Assignments	Nodes	Backtracks
dsdf	28.2479178905	76	72	27
ssdf	37.9557471275	345	239	296
ddod	43.0809731483	345	239	296

Figure 3: Table showing data for the Queen7_7 graph

7 Discussion

From the results shown in Sec. 6, it is shown that for different instances of graph colouring, Dynamic Smallest Domain First, produces the solutions the fastest with the least number of assignments, the smallest number of nodes expanded during search, and the least number of times backtracking was needed. The heuristics involving solely the number of arcs produced the worst results by having the most assignments, the most number of nodes expanded during search, and the most number of times backtracking was needed.

In this practical, all of the basic requirements are met including binary constraints, minimization, and detailed experimentation using a popular problem class. Static ordering variables is allowed by the constraint solver as well as smallest domain first orderings. As an extension, unary constraints are added to the variables and dynamic ordering heuristics are provided.

Rocketbear has shown to be effective at providing solutions to graph colouring using a constraint solver, and has shown that certain heuristics produce solutions faster and less costly than other heuristics.