Automatically Detecting Neighbourhood Constraint Interactions using Comet

Alastair Andrew

University of Strathclyde, Glasgow, G1 1XH, UK
alastair.andrew@cis.strath.ac.uk

Abstract. Recently there has been an interest in easing the implementation of Local Search algorithms through the development of specialised languages and frameworks. There has also been an integration of Constraint Programming ideas into Local Search in the form of the language Comet [1]. This combination of Local Search with constraints allows us to explore whether Local Search neighbourhoods effect constraints in a predictable fashion. We have implemented a system in Comet that can detect which neighbourhoods interact with constraints in the International Timetabling Competition problem instances. The aim is that this interaction information can be used to simplify the creation of multi-phase algorithms or highlight potentially redundant constraints. The results from these preliminary experiments have been encouraging and show the viability of completely automating neighbourhood constraint interaction detection.

1 Introduction

From almost the very beginnings of Computer Science in the 1940s researchers have been fascinated with trying to develop systems which think intelligently. There are many different facets of intelligence but one of the most compelling is the power to reason about the consequences of one's decisions. The Automated Planning community in particular relies heavily on this ability to rationalise the effects of the actions one executes. For traditional Local Search algorithms the effect of exploring a particular neighbourhood is less well defined. The exploration of the search space is guided by an acceptance function which assesses each of the neighbouring solutions based upon their fitness value. Many different strategies exist for implementing these acceptance functions but since the fitness value is typically a single numerical value there is little reasoning that can be done. Solutions which improve on the current fitness are usually accepted. Some meta-heuristics such as Simulated Annealing have the ability to accept non improving solutions in order to achieve greater diversification. We predict that more expressive algorithms could be developed if the search had additional fitness information rather than a single homogenous fitness value.

A recent trend in the Local Search field is the move towards Constraint-Based Local Search spearheaded by Van Hentenryck and Michel with their development of the language Comet. Constraint-Based Local Search is a hybrid of Constraint Programming (CP) and Local Search paradigms

which allows the accurate modelling of a problem whilst providing features which simplify writing different search strategies. Of prime interest to us is the ability to maintain a count of the violations of the problem constraints independent of each other. This additional information can be used to determine whether a solution is actually an improvement. For example if a move is evaluated and increases the total number of constraint violations yet reduces the hard constraint violations, bringing the solution closer to feasibility, should this still be considered a worsening move?

The exploitation of constraint neighbourhood interaction underpins multi-phase algorithms such as Kostuch's winning entry [2] in the first International Timetabling Competition (ITC)[1]. The approach taken was to find a feasible solution which satisfied the hard constraints of the problem and then only explore other feasible solutions when trying to optimise the soft constraints. The choice of neighbourhoods for the optimisation phase guaranteed that the neighbours searched would still be feasible. The selection of these neighbourhoods and the observation that they did not interact with the hard constraints was based upon human intuition and highlights Kostuch's skill as algorithm designer. Our system seeks to aid the algorithm designer by automatically highlighting these constraint neighbourhood relationships thus simplifying the creation of effective multi-phase algorithms. In problems with multiple constraints spotting potentially beneficial interactions is by no means trivial. Our system can also be used to indicate whether a constraint model can be optimised, if the search neighbourhoods do not effect a constraint then, as long as the initial solution satisfies it, the constraint is not required since the search will never be able to introduce a violation. Van Hentenryck and Michel trim out an unnecessary constraint from their Magic Square implementation [1, p. 45] by exploiting their knowledge of how the search will interact with the problem model.

2 Experiment

The problem chosen for experimental analysis was the Post Enrolment Based Course Timetabling problem which formed the second track in the 2nd International Timetabling Competition[2]. This problem was specifically designed to be a fair representation of timetabling problems and contains a variety of hard and soft constraints which are listed in Figure 1.

Most of the neighbourhoods were constructed following the example of Di Gaspero and Schaerf who competed in the first ITC using a system which searched multiple composed neighbourhoods [3]. This work seeks to complement the automatic creation of neighbourhoods with the automatic detection of their effects. When running the experiment we used seven neighbourhoods. The two atomic neighbourhoods, *Move Event Timeslot* and *Move Event Room*, were combined to create the composite neighbourhood *Move Event Room & Timeslot*. Similarly *Swap Event Timeslot*

[1] `http://www.idsia.ch/Files/ttcomp2002/`

[2] `http://www.cs.qub.ac.uk/itc2007/`

- Hard Constraints
  - **c1** No student attends more than one event at the same time.
  - **c2** Events can only be placed in room with sufficient capacity / features.
  - **c3** Only one event can be in a room during a given timeslot.
  - **c4** Events can only be in timeslot pre-defined as available.
  - **c5** If specified, events must be scheduled in the correct order.
- Soft Constraints
  - **c6** Where possible events should not occur in the final timeslot of a day.
  - **c7** Students should not attend more than two consecutive events.
  - **c8** Students should never have only one event on a day.

Fig. 1: The post enrolment based course timetabling problem constraints.

and *Swap Event Room* gave rise to the *Swap Event Room & Timeslot* neighbourhood. The final neighbourhood, *Swap All Event Timeslots*, exchanges all the timeslots of all the events currently placed within the chosen timeslots.

3 Approach

We propose that *simulation* can be used to automatically detected the interactions between problem constraints and the search neighbourhoods. By monitoring the state of the constraint violations before a move is selected and then comparing it with the resulting state we can gain a partial picture of which constraints can be effected. Clearly if selecting a move within a neighbourhood leads to the alteration of the constraint violations then that neighbourhood can effect a change. Our implementation contains some interesting design choices. Firstly we use multiple`ConstraintSystem` objects, one for each of the 8 problem constraints. We then assign the incremental variable of the `ConstraintSystem` violations to an array, `constraintViolations`, which is indexed by an `enum` value unique to that constraint. The neighbourhoods are represented as objects which contain a main method specifying how a move is performed. As well as the main method and other related search methods there is also the capability to pass an `enum` to the `interactsWith()` method which stores relationships in a set.
Comet has an advanced system for the handling of events. Any incremental variable in a Comet program can have an event listener attached. An event listener can trigger a block of code when it catches the event. Comet uses this system primarily to ease the implementation of meta-heuristics such as Tabu Search and also for updating GUIs. In this implementation we used the `@changes()` listener attached to the variable representing the number of violations of each constraint. The simulation code can be seen in Figure 2, of particular interest is the use of the *in* keyword. This essentially limits the scope of the listener meaning that it will only respond to events thrown from within the following block of code. Once the thread of execution has left this block then the listener is discarded. This means that a new listener is instantiated for each neighbourhood constraint pair preventing any chance of an interaction being erroneously attributed to a previous neighbourhood.

```
void detectConstraintInteractions(set{NeighbourhoodMove} nhoods){
    forall(n in nhoods){
        createInitialSolution();
        var{bool} found(solver) := false;
        forall(t in constraintType){
             when constraintViolations[t]@changes(){
                 n.interactsWith(t);
                 found := true;
             } in {
                 n.run();
                 if(!found){
                     n.exploreNeighbourhood();
                     while(n.hasMoves() && !found){
                         call(n.getMoveFromSelector());
                     }
                 }
             }
        }
    }
}
```

Fig. 2: Comet implementation of detection method.

The major benefit of using events as the basis for our detection system is the clean separation between the neighbourhoods and detector which we can achieve. The detector simply iterates over a set of Neighbourhood objects and checks each for interactions. The acceptance function for the neighbourhood is set to accept any fitness. For purposes of detecting an interaction it does not matter whether a move reduces or increases the constraint violations; both indicate that a relationship exists.
The *simulation* is performed in two stages. Starting from a randomly created initial solution a random move from the neighbourhood is chosen, often this will lead to a constraint change and prevent the need for further exploration. For some constraints the chance of randomly selecting a move which would violate it is fairly low and so a more rigourous search is required. If the initial move has not found any interaction then the detector explores every neighbouring state from the current position. If at any stage a change of the constraint violations is detected then the exploration is stopped.

4 Results

The preliminary results presented in Table 1 show that simulation detection is correctly able to establish the relationships in all the cases where a relationship does exist. In situations where no relationship exists there are only two false positive detections, both on constraint *c6*. This highlights one of the main weakness of simulation; the inability to detect cyclical dependencies.

Neighbourhood	c1	c2	c3	c4	c5	c6	c7	c8
1 Move Event Timeslot	1	0	1	1	1	1	1	1
2 Move Event Room	0	1	1	0	0	0	0	0
3 Move Event Room & Timeslot	1	1	1	1	1	1	1	1
4 Swap Event Timeslots	1	0	1	1	1	1	1	1
5 Swap Event Room	0	1	0	0	0	0	0	0
6 Swap Event Room & Timeslot	1	1	0	1	1	1	1	1
7 Swap All Events Timeslots	0	0	0	1	1	1	1	1

Table 1: Results for ITC instance 2007-2-1.tim. The shaded cells indicate where relationships exist. The underlined values highlight incorrect detections.

Frequently a constraint will specify that a variable should never be assigned a particular value. In the ITC the final timeslot constraint, *c6*, is an example. The penalty that violating this constraint accrues is dependant upon the number of students within the incorrectly placed event. If the search procedure exchanges the timeslots of two classes, one in the invalid timeslot, then this will lead to a change in the penalty score. Although the penalty score fluctuates the interchange neighbourhood will never remove this violation and the best that could be achieved is a score equal to the size of the smallest event. The problem can be removed by monitoring the number of violations (rather than the violations weighted by the class size) but at present we have not rectified this issue in our implementation.

5 Conclusions

With this work we have indicated that the process of detecting a neighbourhood's constraint interactions can be performed automatically with reasonable accuracy. We hope that this information will aid the creation of more efficient algorithms which can exploit this hidden structure. A benefit of our approach is that it is loosely coupled to the neighbourhoods and constraints which we feel is in keeping with Van Hentenryck and Michel's desire that "Local Search = Model + Search" [1, p. xiv]. Although the preliminary experiment has been conducted using the ITC problems it contains no timetabling specific elements and should be applicable to any constrained problem. It should be noted that although we have implemented this system in Comet there is no reason why the same approach couldn't be applied to other languages. We make extensive use of Comet's event system which allows us to create quite a succinct method but the cycle of applying moves whilst monitoring the resultant violation changes is not Comet specific.

6 Future Work

This work is still very much in its early stages and there are many issues remaining to be explored. At present there is no caching of the

relationships identified by the simulation. Since all the instances of a given problem will be subject to the same interactions then if the initial detection results are cached future detection can be avoided. Currently the simulation method takes several minutes to detect the interactions between all constraint and neighbourhood pairs which is a considerable overhead for each search run.

We are also exploring the possibility of using reflection to discover constraint neighbourhood relationships. By parsing the constraint model of the problem it is possible to extract which variables the constraints are defined across. If a neighbourhood alters a disjoint set of variables from those involved in a constraint it can safely be assumed that no interaction will take place.

Local Search's effectiveness will always be tightly coupled to the selection of efficient neighbourhoods which allow the traversal between high quality solutions. Guidelines for the design of efficient neighbourhoods are hard to find with most neighbourhoods arising chiefly from human intuition and prior experience. As noted earlier, work on the composibility of neighbourhoods like that of Di Gaspero and Schaerf has sought to automate or, at the very least, partially codify this process. Their development of the EasySyn++ [4] section of the EasyLocal++ framework illustrates further progress in this direction. Other avenues which could be explored are the evolution of neighbourhood structures. Within the Automated Planning community evolution has been successfully used to create useful macro-actions [5] (which are analogous to search neighbourhoods).

References

1. Van Hentenryck, P., Michel, L.: Constraint-Based Local Search. 1st edn. The MIT Press, Cambridge, Massachusetts (2005)
2. Kostuch, P.: The University Course Timetabling Problem with a Three-Phase Approach. In Burke, E.K., Trick, M., eds.: Practice and Theory of Automated Timetabling V. Volume 3616 of Lecture Notes in Computer Science., Heidelberg, Spring Berlin (November 2005) 109–125
3. Di Gaspero, L., Schaerf, A.: Multi-neighbourhood local search with application to course timetabling. In Burke, E.K., De Causmaecker, P., eds.: Practice and Theory of Automated Timetabling IV. Number 2740 in Lecture Notes in Computer Science. Springer-Verlag, Berlin-Heidlberg, Germany (2003) 263–278
4. Di Gaspero, L., Schaerf, A.: EasySyn++: A Tool for Automatic Synthesis of Stochastic Local Search Algorithms. In Stützle, T., Birattari, M., Hoos, H.H., eds.: Engineering Stochastic Local Search Algorithms. Designing, Implementing and Analyzing Effective Heuristics. SLS 2007. Volume 4638 of Lecture Notes in Computer Science., Springer (2007) 177–181
5. Newton, M.A.H., Levine, J., Fox, M., Long, D.: Learning Macro-Actions for Arbitrary Planners and Domains. In: Proceedings of the 17th International Conference on Automated Planning and Scheduling ICAPS 07. (September 2007)