Automatically Detecting Neighbourhood Constraint Interactions using Comet

Alastair Andrew and John Levine

University of Strathclyde, Glasgow, G1 1XH, UK,
`{firstname.lastname}@cis.strath.ac.uk`

Abstract. Local Search has been shown to be capable of producing high quality solutions in a variety of hard constraint and optimisation problems. Typically implementing a Local Search algorithm is done in a problem specific manner. In the last few years a variety of approaches have emerged focussed on easing the implementation and creating a clean separation between the algorithm and problem. We present a system which can deduce information about the interactions between problem constraints and the search neighbourhoods whilst maintaining a loose coupling between these components. We apply this technique to the International Timetabling Competition instances and show an implementation expressed in Comet.

1 Introduction

Over the last ten years there has been a concerted effort by the community to simplify the implementation of Local Search algorithms. This has taken different forms such as adding libraries to existing constraint solvers like the *tentative*[1] library for the ECL^{i}PS^e Constraint Programming System [1]. ParadisEO (specifically the MO module) [2] and EasyLocal++ [3] are frameworks for C++ algorithm development. Some complete Local Search programming languages such as SALSA [4], Localizer [5] and most recently Comet [6] have been created. This trend toward Constraint-Based Local Search is succinctly captured by Van Hentenryck and Michel's belief that "Local Search = Model + Search" [6, p. xiv]. The constraint model of a problem should be separated from the search implementation. This encourages reusable and maintainable algorithms rather than tightly coupled and problem specific solutions.

The most effective Local Search algorithms exploit properties of a problem's structure to help solve them more efficiently. Croes' application of the 2-opt neighbourhood to the Travelling Salesman Problem (TSP) [7] is a classic example of leveraging knowledge about the problem to reduce unnecessary exploration. The aim of the TSP is to create a travel itinerary for a salesman which visits all the cities in the problem instance only once and leads to the shortest possible tour. The tour should be a Hamiltonian cycle, starting and finishing at the same city. By only exploring neighbouring solutions created by the 2-opt operator the

[1] The *tentative* library supersedes the older *repair* library

resultant cycle can be guaranteed to remain Hamiltonian (providing the starting solution was).

In our terminology we would say that the 2-opt neighbourhood does not *interact* with the constraint that the tour is Hamiltonian. A neighbourhood which does *interact* with a constraint is one which can cause a change in the constraint violations. We are not concerned whether the change is positive or negative only that it occurs. The rationale behind this is that in the context of Local Search the neighbourhood is responsible only for presenting the Acceptance Function with different candidate solutions to consider. It is up to the Acceptance Function to decide which of the these candidates it will select and this does not always have be one which reduces the constraint violations.

Restricting the exploration to solely feasible solutions for the TSP is trivial but for problems with more constraints this is unlikely to be the case. Multi-phase or multi-stage algorithms take the approach of Croes further. Typically the first phase of the algorithm searches solutions attempting to find a feasible assignment which adheres to the problem's hard constraints. Upon finding a solution which satisfies the hard constraints the algorithm will switch into an optimisation phase where it attempts to improve the objective function or satisfy the soft constraints whilst exploring only feasible solutions. This multi-phase approach has been applied successfully in both the International Timetabling Competitions (ITC)[2][3]. The best performing entries in the original competition employed a multi-phase approach [8, 9]. The winning entrant of first and third tracks of the most recent ITC also employed a multi-stage approach [10]. In Müller's solver after the initial constructive phase there is a Hill Climbing phase where the hard constraints are strictly adhered to. The second placed entrants of track 1, Gogos et al [11] have a constructive phase followed by a Local Search phase which uses neighbourhoods based on Kempe Chains to maintain feasibility. The winners of the 2nd track, Cambazard et al [12] use a hybridisation of Constraint Programming with sections of Local Search and again during the optimisation phase the search is restricted to solely feasible solutions.

Neighbourhood constraint interaction is useful in other situations such as refining the constraint model of a problem. In Van Hentenryck and Michel's implementation of the Magic Square problem [6, p. 45] they comment that they have omitted explicitly stating the constraint that all the values should be allocated to a position in the square. Their initial solution is created such that it satisfies this constraint and then their search procedure only explores swaps (thus maintaining feasibility). In this situation they have used their knowledge of how the neighbourhood effects the problem constraints to remove a redundant constraint from the model.

Designing multi-phase algorithms and removing potentially redundant constraints from problem models are two examples of situations where human intuition regarding the interplay between the search neighbourhoods and the problem constraints is required. If we are to achieve the "Local Search = Model +

[2] 1st ITC `www.idsia.ch/Files/ttcomp2002/`
[3] 2nd ITC `www.cs.qub.ac.uk/itc2007/`

Search" ideal then detecting these *interactions* automatically would be beneficial to further reduce the coupling between the constraint model and search procedure. Detection can provide extra information about the underlying structure of a problem which may not be immediately apparent to a human.

2 Experiment

As mentioned previously multi-phase algorithms have proven very successful in the ITC so we elected to use the Post Enrolment Based Course Timetabling version as the experimental testbed. This problem comprised the second track of the most recent ITC and was essentially a slightly richer version of the original ITC problem. The problem was designed to be representative of timetabling problems and contains eight constraints, five of which have been designated as hard constraints. Figure 1 shows these constraints and the abbreviations which will be used in the remainder of the paper. It is also worth noting that when we are discussing constraints we are referring to the general class of a constraint not the specific instance. To illustrate this consider the constraints "Event 1's room is not Room 1" and "Event 10's room is not Room 2", each of these situations will be instantiated when stating the model, however we would treat both these as constraint **c3**. The work of Ågren [13] has explored the relationship between constraints and neighbourhoods but in the context of using individual constraints to suggest appropriate neighbourhoods rather than generalising for all constraints of a type. In Ågren's *Constraint-Directed Neighbourhoods* each individual constraint instance automatically creates neighbourhoods which can preserve, increase or decrease the constraint penalty. Our work attempts to apply the notion of neighbourhood constraint interaction to all the constraints from a general class and uses existing neighbourhoods rather than explicitly deriving neighbourhoods from the constraints.

- 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** Events shouldn't where possible 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.

To study the interactions between these constraints and search neighbourhoods we also need to define a set of neighbourhoods. The neighbourhoods we chose to develop were based upon those used by Di Gaspero and Schaerf in their entry to the original timetabling competition [14]. Their work sought to structure the creation of neighbourhoods by allowing larger neighbourhoods to be composed of smaller atomic neighbourhoods. For the timetabling problem the two most basic neighbourhoods are *Move Event Timeslot* and *Move Event Room* which assign an event to a new timeslot or room respectively. Our implementation of *Move Event Room* can be found in Listing 1.1. Both these move neighbourhoods can be combined to give *Move Event Room & Timeslot* which assigns an event to a new room and timeslot. Note that it will not consider situations where only one of the two assignments has actually changed as the atomic neighbourhoods would have already discovered these solutions.

The next three neighbourhoods we have implemented, *Swap Event Timeslot*, *Swap Event Room*, *Swap Event Room & Timeslot*, are versions of the Move neighbourhoods but generalised to be exchanges. In these neighbourhoods two events are selected and their values are swapped. Only situations where these swaps will lead to a change are considered so for example *Swap Event Room* will not select two events which are in the same room.

The final neighbourhood in our system is *Swap All Events Timeslots*, this neighbourhood selects two timeslots and then exchanges all the events occurring at those times (whilst leaving the room assignments fixed).

```
forall(e in Events, r in Rooms: r != eventRoom[e]){
    neighbor(S.getAssignDelta(eventRoom[e],r), N){
        eventRoom[e] := r;
    }
}
```

Listing 1.1. Comet implementation of Move Event Room

The experiment was implemented in Comet[4]. Several factors contributed to this choice, primarily it was Comet's support for differentiable objects which greatly simplifies the development of incremental algorithms. The evaluation of the fitness of neighbouring solutions is one of the most computationally expensive parts of Local Search, depending on the Acceptance Strategy being used each neighbourhood may need to be exhaustively expanded to find the best possible neighbour. Based upon the observation that neighbouring solutions will share a lot of commonality, incremental algorithms aim at calculating only what has changed between these solutions. This removes the need to evaluate each solution from scratch which is costly. However implementing incremental algorithms is not simple (and is tightly coupled to the neighbourhood moves) so any system which removes this distraction and allows us to concentrate on the interaction detection is a bonus. Besides the support for differentiable objects Comet also

[4] `http://www.comet-online.org`

has an advanced event handling system [6, p. 156]. Events can be attached to variables so that whenever a change occurs a listener is informed and can perform some operations. Previously this has been used to keep visualisations updated or to control meta-heuristics (flagging when iteration limits have been reached or a Tabu tenure expires etc.).

3 Detection by Simulation

Our proposed technique for detecting neighbourhood constraint interactions is one we have dubbed *simulation*. At its core is the idea that easiest way to find out whether a neighbourhood interacts with a constraint is by exploring the neighbourhood. By monitoring the violations of the constraint under inspection whilst exploring we can detect if a relationship exists. If at any point during the exploration the violations change then we have confirmation of a relationship and can move onto exploring other neighbourhood / constraint combinations.

To actually implement this we need a few minor changes to a conventional constraint model. In Comet models, an object called the ConstraintSystem is used to define the constraints. This object is responsible for maintaining information about the current constraint violations and which of the problem variables are responsible. Normally there is only a single instance of the ConstraintSystem which manages all the constraints in the problem. For our system to work we need to be able to track the violations of each of the eight problem constraints so we need to create eight ConstraintSystems. Each constraint, **c1**–**c8**, is posted to a separate ConstraintSystem.

The simulation code can be found in Listing 1.2. There are various aspects to note. Firstly there is no stipulation where the neighbourhoods come from, they are passed into the detector as a set. The detector works by iterating through each of the neighbourhoods and checking for interactions with each of the problem constraints. To ease referencing to the constraint types the variables maintaining the violations are stored within an array which is indexed by an **enum** value of the constraint name. When an interaction is detected the **enum** of the constraint is passed into the neighbourhood which maintains a set of the constraints it effects.

The use of the **when** / **in** block, lines 6 and 9 respectively, is important for efficiency and correctness. The **in** keyword limits the scope of the event listener, **@**changes(), to only the code in the block following the **in**. Once the thread of execution has left this block the listener will be discarded, this prevents multiple listeners still being active and consuming excess resources and it also guards against multiple listeners being alerted and interactions being attributed to the wrong neighbourhoods.

The neighbourhoods themselves are all children of a NeighbourhoodMove class which provides them with common functionality such as being able to store the **enum**s of the constraints they interact with. It also provides the methods run() and exploreNeighbourhood(), the former randomly accepts a move from within

```
void findConstraintInteractions(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());
                    }
                }
            }
        }
    }
}
```

Listing 1.2. Comet implementation of detection method.

the neighbourhood and the latter exhaustively explores the entire neighbourhood if the random move didn't trigger the event listener.

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

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	$\underline{1}$	▨ 1	▨ 1
5 Swap Event Room	0	▨ 1	0	0	0	0	0	0
6 Swap Event Room & Timeslot	▨ 1	▨ 1	0	▨ 1	▨ 1	$\underline{1}$	▨ 1	▨ 1
7 Swap All Events Timeslots	0	0	0	▨ 1	▨ 1	▨ 1	▨ 1	▨ 1

Table 1 shows the accuracy of the simulation method. As can be seen (denoted by the underlined values) there are only two false positive results when the simulation indicates that *Swap Event Timeslot* & *Swap Event Room & Timeslot* interact with constraint *c6*. This highlights the simulation method's main weakness, its inabililty to detect cyclical exchanges. The constraint in question states that no event should be placed within the final timeslot of a day, obviously any neighbourhood which is swapping event's timeslot allocations will always be placing a new event into the final timeslot space. The reason the simulation detects this as being an interaction is due to the evaluation of the particular constraint. In the ITC domain objective function the violations of this constraint are multiplied by the number of students effected by the violation. The aim is to penalise solutions which have large events placed into the final timeslot. As the simulation only monitors the weighted violations score when checking for interactions when two events with different numbers of students attached to them are exchanged then the score will alter and it will be flagged as an interaction. We could rectify this problem by monitoring the unweighted number of constraint violations rather the penalised version but at present we have not included this.

The other main drawback is that whilst it is easy to capture when a relationship exists, because Local Search is an incomplete search strategy you cannot conclusively state that a relationship *does not* exist. Currently if after an exhaustive exploration of a neighbourhood no interactions are detected the detector assumes that no interaction exists.

It is worth noting that whilst we have implemented this simulation system within Comet there is nothing which could not be replicated in other languages. We do exploit the event system Comet provides to make the simulation code more concise but it is certainly not infeasible to replicate this variable monitoring in alternative packages.

When the detection is finished it outputs the results as a constraints by neighbourhoods plain text file containing a matrix of interactions. The primary reason we do this is because the information extracted by this process should be applicable to all instances of a problem so only one detection run is required. Running the simulation detection method can take a few minutes and as the number of constraints or neighbourhoods grows then so does the time which is in the order of $\mathcal{O}(n^2)$.

4 Conclusions

This work is still in its preliminary stages but shows promise that the interactions between search neighbourhoods and problem constraints can be discovered with reasonable accuracy by a general system which requires no problem specific knowledge. When applied to the ITC instance the simulation method was correctly able to identify all the constraint interactions. The false positive interactions it highlights are due to the use of weighted constraint violations and in future versions this flaw will be eliminated. Neighbourhood constraint interaction information can be used to refine constraint models removing unneeded constraints or to aid in the development of multi-phase algorithms. The goal is not to replace the role of the human algorithm designer, simply to provide more information and possibly highlight interactions that they may have overlooked. We feel this is in keeping with the direction of the community and helps to simplify the design Local Search algorithms.

5 Ongoing and Future Work

We are currently exploring several different directions to expand this work. The furthest developed is an alternative detection technique which we have named *reflection* after the process by which a programming language can examine its own code. This feature is common in many established languages such as SmallTalk, Java (using the java.lang.reflect package) and Python. Our notion of reflection is an analysis of the source code performed at runtime. Simulation is effective at detecting interactions where they exist but cannot quickly discount situations where no interaction occurs.

```
1  include ‘‘LocalSolver’’;
2  range r = 1..10;
3  LocalSolver ls();
4  var{int} x(ls,r) := 10;
5  var{int} y(ls,r) := 10;
6  ConstraintSystem S(ls);
7  S.post(x < 5);
8  ls.close();
9
10 MinNeighborSelector N();
11 forall(i in y.getDomain()){
12     neighbor(S.getAssignDelta(y,i),N){
13         y := i;
14     }
15 }
16 if(N.hasMove()){
17     call(N.getMove());
18 }
```

Listing 1.3. An example problem highlighting an unrelated constraint and neighbourhood.

In Listing 1.3 there is a simple example which highlights this problem. The problem has two variables only one of which, **int** x, is actually used to define the constraint. The second part of the program contains the neighbourhood definition. The neighbourhood is making an assignment to the variable y (at line 13) which is not part of the ConstraintSystem. Simulation would exhaustively explore all the states generated by this neighbourhood before concluding that no relationship existed. If the set of variables which are present in the constraint and neighbourhood were compared they would be disjoint ruling out the chance of interaction. At present we can extract the lines in the source file where neighbourhood statements are. In Comet these are quite easy to spot because the assignment operators **:=** and **:=:** must appear. The **neighbor** keyword is another obvious trait which can be used to identify neighbourhoods. The parser is not completely finished and cannot at present identify all the variables or their scopes. The goal is ultimately to provide this reflection capability in a Comet library which only requires one additional include statement in your code to utilise much like Comet's existing transparent parallel search feature [15].

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 composability 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++ [16] 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 [17]. Macro-actions are composite operators which alter the way which the search space can be traversed and can be thought of as analogous to Local Search neighbourhoods. In such scenarios then our detection scheme could act as a fitness guide for created neighbourhoods, our intuition being that it is desirable to create neighbourhoods which interact with as few constraints as possible.

References

1. Apt, K.R., Wallace, M.: Constraint Logic Programming using ECL^{i}PS^e. 1st edn. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK (2007)
2. Cahon, S., Melab, N., Talbi, E.G.: ParadisEO: A Framework for the Reusable Design of Parallel and Distributed Metaheuristics. Journal of Heuristics **10**(3) (May 2004) 357–380
3. Di Gaspero, L., Schaerf, A.: EasyLocal++: an object-oriented framework for flexible design of local search algorithms. Software – Practise & Experience **33**(8) (July 2003) 733–765
4. Laburthe, F., Caseau, Y.: SALSA: A Language for Search Algorithms. Constraints **7**(3–4) (July 2002) 255–288
5. Michel, L., Van Hentenryck, P.: Localizer. Constraints **5**(1–2) (January 2000) 43–84
6. Van Hentenryck, P., Michel, L.: Constraint-Based Local Search. 1st edn. The MIT Press, Cambridge, Massachusetts (2005)
7. Croes, G.A.: A Method For Solving Traveling-Salesman Problems. Operations Research **6**(6) (November–December 1958) 791–812
8. Chiarandini, M., Birattari, M., Socha, K., Rossi-Doria, O.: An effective hybrid algorithm for university course timetabling. Journal of Scheduling **9**(5) (October 2006) 403–432
9. Socha, K.: $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System for International Timetabling Competition. Technical Report CP 194/6, IRIDIA, Université Libre de Bruxelles, Av. Franklin D. Roosevelt 50, 1050 Bruxelles, Belgium (March 31 2003)
10. Müller, T.: ITC2007: Solver Description. Technical report, Purdue University, West Lafayette IN 47907, USA (2008)

11. Gogos, C., Alefragis, P., Housos, E.: A Multi-Staged Algorithmic Process for the Solution of the Examination Timetabling Problem. Technical report, Department of Electrical and Computer Engineering, University of Patras-Greece (2008)
12. Cambazard, H., Hebrard, E., O'Sullivan, B., Papadopoulos, A.: Local Search and Constraint Programming for the Post Enrolment-based Course Timetabling Problem. In Gendreau, M., Burke, E.K., eds.: Proceedings of the 7th International Conference on the Practice and Theory of Automated Timetabling (PATAT 2008), Springer (August 2008)
13. Ågren, M.: Set Constraints for Local Search. PhD thesis, Department of Information Technology, Uppsala University, Sweden (December 2007)
14. 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
15. Michel, L., See, A., Van Hentenryck, P.: Parallelizing Constraint Programs Transparently. In Bessiere, C., ed.: Proceedings of the 13th International Conference on the Principles and Practice of Constraint Programming - CP 2007. Volume 4741 of Lecture Notes in Computer Science., Providence, Rhode Island, Springer-Verlag (September 2007)
16. 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
17. Newton, M.A.H., Levine, J., Fox, M., Long, D.: Learning Macro-Actions for Arbitrary Planners and Domains. In Boddy, M., Fox, M., Thiébaux, S., eds.: Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS 07), Providence, Rhode Island, USA, AAAI Press (September 2007) 256–263