Constraint Directed Variable Neighbourhood Search

Alastair Andrew, John Levine, and Derek Long

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

Abstract. Local search algorithms operate by making small changes to candidate solutions with the aim of reaching new and improved solutions. The problem is that often the search will become trapped at sub optimal states from where there are no improving neighbours. Much research has gone into creating schemes to avoid these local optima and various strategies exist mainly based around altering the acceptance function. Another approach is Variable Neighbourhood Search which aims to bypass optima by linearly switching through multiple search neighbourhoods. We propose a new method where the selection of neighbourhoods is dynamically decided dependant on the violations of the problem constraints, Constraint Directed Variable Neighbourhood Search. We compared Constraint Directed Variable Neighbourhood Search to Variable Neighbourhood Search and show that the same search progress can be achieved whilst exploring only a fraction of the states.

1 Introduction

All problems in the real world are subject to some form of constraints on their possible solutions. Without constraints then there would be no way to differentiate between valid solutions and those which are infeasible. Scheduling manufacturing processes, timetabling exams, and rostering employees' shift patterns are common, heavily constrained problems. The laws of physics are surprisingly inflexible and any attempt to schedule two jobs to occur simultaneously on the same machine is destined to fail. Not all constraints are impossible to violate, but it may be undesirable; few employees would be happy to find they had been rostered to work for 24 consecutive hours!

Local search is one technique which has been successfully applied to numerous hard constrained problems. It is capable of producing high quality solutions within acceptable time limits where other techniques such as Constraint Programming and Linear Programming struggle. Perhaps one of its greatest strengths is that it retains an appealing conceptual simplicity. The basic idea is that by making small changes to a series of improving solutions the global optima can be reached. Unfortunately this is rarely the case and one of the main weaknesses of local search is its propensity to become trapped at local optima. These are states which appear to be better than all the surrounding solutions but are not the global optima. Much research has gone into allowing local search to

escape from, or avoid becoming trapped at, these local optima. Meta-heuristics such as Simulated Annealing and Tabu Search advocate manipulating the acceptance function. Solutions may be accepted which are not strictly better than the current solution in the hope that the search will explore new areas and avoid becoming trapped. Iterated Local Search takes the local optima and then applies some perturbation to it to create a new solution which is hopefully sufficiently different to allow the search to progress in a more fruitful direction.

In the late 1990s Mladenović and Hansen [1] put forward a new framework for local search algorithms, Variable Neighbourhood Search (VNS). They take a different perspective to the problem of local optima; while there may be no improving states in a current neighbourhood a different neighbourhood may allow the search to progress to better solutions. Local optima are specific to neighbourhoods and so by exploring more neighbourhoods the likelihood of converging towards the global optima increases.

In the canonical form of VNS the order in which the neighbourhoods are explored is defined by the algorithm designer. The most common configuration is for the neighbourhoods to be explored in ascending order of size. This gives a good trade off between intensification and diversification. Once an improving solution is selected VNS usually restarts from the first neighbourhood in its linear sequence.

There have been some interesting developments applied to the Unit Commitment problem by Viana, de Sousa and Matos [2]. Viana et al make the assertion that certain neighbourhood moves are hard to recover from, that is to say they leave the solution in an infeasible state that requires several other moves to reestablish feasibility. They attempt to precompute chains of potential neighbourhood moves which can quickly return a solution to feasibility. The neighbourhood chains they apply are determined by the violations of the problem's constraints.

We propose a new strategy where the neighbourhoods are varied dynamically depending upon the number of violations of the problem constraints. With Constraint Directed Variable Neighbourhood Search (CDVNS) we aim to show that by using information about the constraint violations then only a subset of the neighbourhoods need to be explored to reach the same goal states.

2 The Problem

To perform some experimentation a suitable problem domain was required. The Bin Packing Problem seemed a good choice. In its basic form the problem is easy to understand and implement. There was also a large number of problem instances available for use. This work uses instances from the Operations Research Library [3].

The basic premise of the Bin Packing Problem is very simple. There are N packages of various sizes, W_i, which need to be allocated to a series of bins which have a fixed capacity C. The problem is subject to a single constraint, namely

bins cannot be filled beyond their capacity. The object of the problem is try and assign the packages into as few bins as possible.

As the focus of this work was into the interaction between the problem constraints and local search neighbourhoods we decided to add several additional constraints to the problem. These constraints were designed to be representative of properties commonly found in other constrained problems. They were also chosen to explicitly guide the search towards sensible solutions. Although the problem instances are for the Bin Packing Problem we treated them as constraint satisfaction problems. Rather than trying to reduce the number of bins used we were attempting to take a solution created by an uninformed greedy heuristic and remove all the constraint violations from it. Another addition we made to the basic problem structure is that packages must be assign to a position within a bin. The number of positions within the bins was fixed as the number of smallest packages from an instance that a bin could feasibly accommodate.

Clashing Packages Constraint For this constraint we split the packages into two different categories based on whether their id's were odd or even. The constraint specifies that all the packages in a bin should be of the same type. Only neighbourhoods which add or remove packages from bins can affect this constraint. If the contents of a bin are left unchanged this constraint cannot be affected. Constraints like this occur in manufacturing scenarios where it is common to find machines which cannot be assigned different types of jobs.

Inter-Bin Constraint The Inter-Bin Constraint states that a bin must be as full as, or more full than any of the following bins. This constraint means that optimal solutions will have the bins arranged in order of total content amount. Often preferences like this are built into the evaluation function of an algorithm but by explicitly stating it as a constraint we are providing richer model of the problem. The most efficient neighbourhood for solving this constraint is one which swaps all the contents of two bins.

Package Ordering Constraint The final additional constraint states that the packages inside a bin must be ordered so that the larger packages appear at lower positions than smaller ones. This constraint is an example of one which is entirely internal to a bin's contents. The most appropriate neighbourhoods are ones which only manipulate the positions of the packages within a bin.

3 Neighbourhoods

We defined twelve different neighbourhoods for our search to explore. Whilst we manually implemented them for this work the eventual goal is that the neighbourhoods will be automatically generated from a formal specification of the problem. Thus while designing the neighbourhoods we tried to make them similar to those we could envisage being inferred. In Di Gaspero and Schaerf [4] they

outline a system for composing neighbourhoods for the timetabling problem. The neighbourhoods can be combined using conventional set notation operators; more complex neighbourhoods are created from basic atomic components.

A recent technical paper by Ågren, Flener and Pearson [5] takes problems specified in Constraint Satisfaction Problem (CSP) notation with the goal of automatically inferring incremental algorithms. Their work focuses on maintaining a complex set representation of the constraint violations which can be used to predict the potential effect of altering a variables assignment. Our work differs from this in that it is more concerned with the interplay between the neighbourhoods and the constraints, however their adoption of CSP formalism is interesting and one which we intend to look into in future work.

A solution to the problem is a list of packages with the bin they have been placed in and the position they occupy within that bin. The most basic possible neighbourhoods would be to alter either the bin or position of a single package. These two neighbourhoods can be augmented by a third neighbourhood in which both the position and bin of a package must be altered. We decided against including a neighbourhood which was the result of a logical union since all the possible solutions could be reached via the other three move neighbourhoods.

The decision to split what could feasibly be a single move neighbourhood which allowed a package have it's bin or position altered into three separate neighbourhoods was intentional. By searching all three neighbourhoods then any possible assignment from the general move neighbourhood could be found so we have not sacrificed any reachability. The benefit is we now have neighbourhoods with guaranteed properties which we can exploit. In addition the three resulting neighbourhoods are all smaller and can be explored more quickly than one more general move neighbourhood.

The next step up from just altering a single package was to extend the three basic neighbourhoods so that they exchanged the assignments of a pair of packages. Neighbourhoods which swap assignments in this fashion have been applied in problem types such the Travelling Salesman Problem, Timetabling and Scheduling. The next abstraction of the initial atomic neighbourhoods was to group together all the packages which were assigned in the same bin. Moving packages as complete groups has the benefit of maintaining any internal relationships those packages may have. The final group of neighbourhoods were created by taking the previous group moves and making them into group swaps. The twelve neighbourhoods presented are not designed to be exhaustive, merely representative of those which could potentially be inferred.

MoveBin Assign a package to a new bin.
MovePosition Assign a package to a new position within the same bin.
MoveBinAndPosition Assign a package to a new bin and location.
SwapBin Exchange two package's bin values.
SwapPosition Exchange two package's position values.
SwapBinAndPosition Exchange two package's bin and positions.
MoveAllBin Move all the packages in one bin to a new bin.
MoveAllPosition Move all the packages in a bin to new positions in that bin.

MoveAllBinAndPosition Move all the packages to new positions in a new bin.
SwapAllBin Swap all the contents between two bins preserving orderings.
SwapAllPosition Swap the positions of packages within the same bin.
SwapAllBinAndPosition Swap all the positions of all the packages in two bins.

4 Constraint Direction

The goal of CDVNS is the intelligent exploitation of the interaction between neighbourhoods and constraints to create a more efficient and stable search progress. Neighbourhoods allow the search to traverse between potential solutions in the state space, crucially though they perform these traversals in a predictable manner. The SwapAllBin neighbourhood maintains the internal ordering of packages within a bin, since the Package Ordering Constraint can only be violated by a change in packages' relative positions any search within the SwapAllBin neighbourhood is guaranteed to neither violate nor satisfy that particular constraint. By examining the current level of constraint violations the aim is to pick the neighbourhood which can reduce the target constraint's violations whilst leaving as many as possible of the other constraints unaffected.

Algorithm 1 Constraint Directed Variable Neighbourhood Search

```
1: while violations > 0 and iterations < limit do
2:     currentConstraint ← GetDominantConstraint
3:     neighbourhoods[] ← neighbourhoodMatrix[currentConstraint]
4:     for all n in neighbourhoods[] do
5:         currentScore ← explore(n)
6:         if currentScore < bestScore then
7:             bestScore ← currentScore
8:         end if
9:     end for
10:    violations ← bestScore.violations
11: end while
```

In any given situation the algorithm must be able to select the most appropriate set of neighbourhoods, this is achieved by storing the possible choices in a matrix structure which is indexed by most pressing constraint. Wherever possible the matrix will return the most specific neighbourhood, however it may be that several neighbourhoods need to be explored. It should be noted that if more than one neighbourhood is returned from the matrix then CDVNS explores them exactly as a conventional VNS would.

The decision about which neighbourhoods should be associated with which constraints is crucial to the performance of CDVNS. If all the available neighbourhoods are associated with every constraint then there would be no perfor-

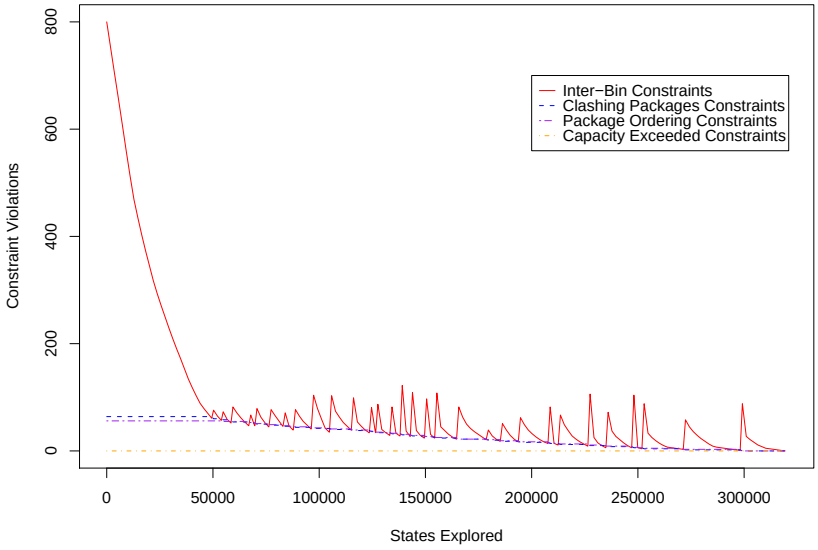
mance advantage over VNS. CDVNS reduces it's exploration by only exploring neighbourhoods which can reduce the number of violations of the dominant constraint. Ideally each constraint type will have only one neighbourhood to search. The best neighbourhood to select is the one which can alter the desired constraint and yet does not impinge on the other constraint types. Unfortunately this clean separation is unlikely to be achievable for all constraints so the selection of neighbourhoods which affects more than just the desired constraint may be unavoidable. The disruption using a less specific neighbourhood can cause to the search progress is explored in more detail in section 5.

Algorithm 1 gives the structure of the CDVNS algorithm. The *currentScore* and *bestScore* variables are objects which contain information about the number of violations of each type of constraint. The *violations* variable is the summation of the number of violations of each of the constraint types. When this value reaches zero then the search terminates as it has found a solution. The *GETDOMINANTCONSTRAINT* method evaluates the *bestScore* and returns which of the constraint types has been violated the largest number of times. This *currentConstraint* is used to access the *neighbourhoodMatrix* which stores the associations between constraints and neighbourhoods. The *neighbourhoods[]* are explored and the best state is selected. In the experiment the *bestScore* is defined as the one which reduces the target violations. An interesting point to note is that currently the implementation of CDVNS is entirely deterministic.

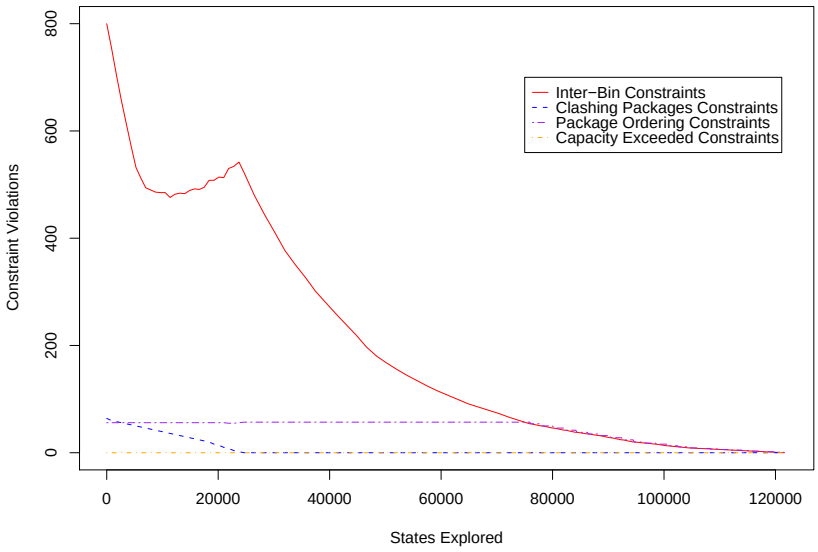
5 Selecting the Dominant Constraint

In the original concept for this algorithm the idea was that the next neighbourhood to be explored would simply be chosen to affect the constraint which had the largest number of violations. During our experimentation it became apparent that this approach was too naive and caused the search to explore more states than was strictly necessary. This can be seen in Figure 1(a) where the search is selecting neighbourhoods based solely upon the amount of violations. Initially the Inter-Bin Constraint violations are dominant and so the search focuses on reducing those. A steady descent of the violations is witnessed until the point where the Clashing Package Constraint violations becomes greater. The neighbourhoods used to solve this particular constraint whilst reducing the intended violations have the side effect of introducing more Inter-Bin Constraint violations. This results in the search focusing on reducing the Inter-Bin Constraints once more until another constraint becomes more violated. This constant disruption of existing constraint violations is counter productive and should be avoided where possible.

To combat this we artificially weighted the Clashing Packages Constraint violations so that the search would always attempt to reduce these before resolving the other constraints. The weighting essentially multiplied the number of Clashing Packages Constraint violations by a thousand so that even a single violation of this constraint would be chosen over hundreds of violations of the other constraints. Figure 1(b) shows the effect of this decision, search progress



(a) Unweighted Constraint Violations



(b) Weighted Constraint Violations

Fig. 1. The effect of constraint weighting on search progress.

becomes much smoother and less than half the amount of state exploration is required. The remaining constraints are selected using the original method. Since the neighbourhoods used to solve the Inter-Bin Constraints and the Package Ordering Constraints do not interact with each other they can safely be interleaved without disrupting the solution. The issue of automatically resolving the order constraints should be tackled in is something we intend to explore in future work. The weighting obviously has limitations, if another constraint happens to be violated more than the arbitrary value which we have selected then the search may still choose to tackle the constraints in a inefficient manner.

6 Initial Results

The initial results have been encouraging Tables 2 & 3 show that the CDVNS algorithm performs better on all the instances. This can be more clearly seen in the graphs, Figures 2(a) & 2(b). We ran the experiment with four different configurations; two acceptance strategies, Best Improvement (BI) and First Improvement (FI), were trialed with both CDVNS and VNS. Each configuration was started from the same initial solution for each instance and we measured the number of solutions explored before reaching one with no constraint violations.

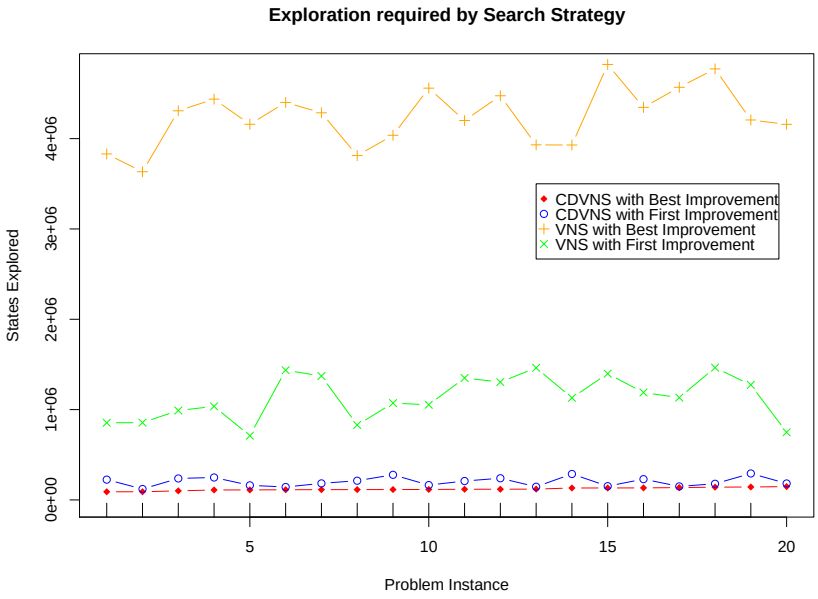
The results are described in terms of state exploration as the objective of CDVNS was to reduce unnecessary exploration. During the experimental runs we did not gather any accurate run time readings, something which we intend to rectify in future experiments. Informally the run times did exhibit differences of similar magnitudes to the state exploration but without proper timing data we cannot make any definitive claims regarding this.

On all the problem instances CDVNS with the BI strategy was the most efficient. Initially this seemed counter intuitive since the BI strategy commits to a complete traversal of each neighbourhood whereas FI only needs to explore until a better state is found. After looking closely at some individual runs it became apparent what was happening was that during the final stages of the search when the Inter-Bin Constraints were being resolved the FI strategy would always choose small swaps. This decision proves to be quite inefficient as an incorrectly placed bin requires several swaps to reach the right position; in effect what happens is a Bubble Sort.

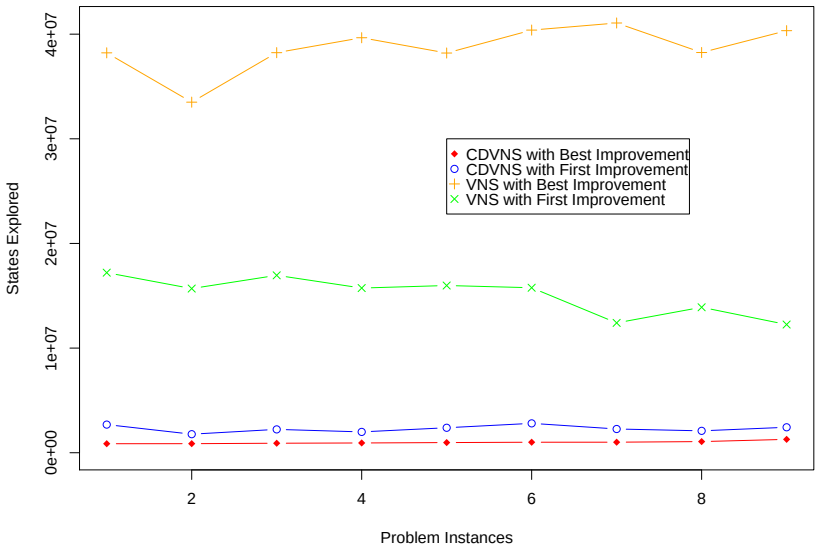
Another interesting point to note is that the performance of CDVNS with BI is the most stable over all the instances. In Table 1 the amount of standard deviation, σ, is far less than the other configurations. Indeed the variation of exploration between the instances in the the VNS strategies is often greater than the total amount of exploration CDVNS required to solve the problem.

7 Conclusions

We have created a variable neighbourhood search algorithm which explores only a subset of the possible neighbourhoods. This subset is determined dynamically during the search based upon how many of the problem constraints remain



(a) 120 Package Problems



(b) 250 Package Problems

Fig. 2. Comparison of exploration required by search.

Configurations		Mean	σ	Min	Q1	Median	Q3	Max
120	CDVNS & BI	119800.0	16223.7	90220.0	112000.0	117000.0	132500.0	147400.0
	CDVNS & FI	202200.0	51532.7	120800.0	159300.0	196300.0	238200.0	292900.0
	VNS & BI	4244000.0	319112.5	3634000.0	4010000.0	4246000.0	4447000.0	4821000.0
	VNS & FI	1133000.0	245937.8	711300.0	956300.0	1132000.0	1355000.0	1465000.0
250	CDVNS & BI	988100.0	128223.1	860900.0	907600.0	970800.0	1007000.0	1278000.0
	CDVNS & FI	2297000.0	327170.7	1773000.0	2093000.0	2269000.0	2433000.0	2807000.0
	VNS & BI	38650000.0	2234899.0	33500000.0	38210000.0	38250000.0	40330000.0	41070000.0
	VNS & FI	15100000.0	1825347.0	12250000.0	13900000.0	15740000.0	15980000.0	17210000.0

Table 1. Descriptive statistics of state exploration.

Instance	CDVNS & BI	CDVNS & FI	VNS & BI	VNS & FI
00	113849	183162	4286047	1371544
01	115860	164363	4558652	1053418
02	147417	182228	4158136	748909
03	118154	209396	4200416	1349938
04	141418	178844	4771796	1464504
05	119660	145849	3930901	1462870
06	133351	230074	4346024	1188831
07	90252	120817	3633546	855290
08	100228	237652	4308257	989947
09	136341	149621	4568430	1132716
10	118425	239959	4474598	1304347
11	143026	292911	4206591	1274017
12	132214	152391	4820972	1397905
13	112440	142975	4400759	1436067
14	110720	161539	4158900	711343
15	110248	248441	4437433	1035583
16	114954	278210	4036979	1071884
17	132165	287235	3929229	1130588
18	114414	212831	3812641	829838
19	90216	224866	3830452	854105

Table 2. State exploration required on the 120 package problem instances.

Instance	CDVNS & BI	CDVNS & FI	VNS & BI	VNS & FI
00	933302	1991219	39658678	15737363
01	1064895	2092704	38245873	13901122
02	970809	2387260	38197133	15979864
03	865854	1773027	33501055	15688430
04	907561	2231076	38231370	16954223
16	1278119	2432534	40333650	12250679
17	1006958	2269030	41066676	12408058
18	1004129	2807196	40388150	15767066
19	860916	2690185	38214876	17207818

Table 3. Partial results of state exploration from the 250 package problem instances.

violated. This approach has been contrasted against an uninformed VNS and we achieved the same search reachability whilst using far less exploration. In addition the performance of CDVNS is more uniform across problem instances.

We have achieved these results whilst retaining the generality of local search, this technique should be applicable to any constrained problem. As our contribution is focused around intelligent neighbourhood selection no stipulations are made about the type of acceptance function used. CDVNS should be compatible with successful meta-heuristics such as Simulated Annealing and Tabu Search.

8 Future Work

This is very much work in progress and there are various directions we wish to explore further. At present the assignment of the weightings which specify the relative importance of the different constraint violations is done manually. We want to investigate whether this can be done automatically by creating a directed graph structure which represents the interactions between constraints. Standard graph labelling techniques should be applicable to resolve the ordering. In more complex problems it is unlikely that we will be able to fully order the constraints as there are liable to be cycles within the graph. However even a partial ordering should still be able to guide the search more efficiently than a static VNS.

Applying some formalism to the problem specification so we can automate the neighbourhood generation is an interesting challenge. This is more in line with Constraint Programming where the focus is on modelling the problem rather than defining how to solve it. To be able to automatically generate the constraint orderings we will also need to investigate how to go about creating the neighbourhood matrix and capturing the interactions between neighbourhoods and constraints.

We are also looking to use the Comet language (Van Hentenryck and Michel) [6], all our current work is implemented in Python. Switching to Comet would give several benefits, namely a cleaner separation between the algorithm and the

problem model and access to high performance delta calculation methods. Our present evaluation function exhaustively recalculates the constraint violations for each state which is a laborious and time consuming endeavour. The separation between the problem and algorithm should make it easy to apply CDVNS to wide range of constrained problems.

We also propose that the interaction between neighbourhoods and constraints can be harnessed to make more efficient delta calculation methods. As we know which constraints cannot be affected by a move within a neighbourhood there will be no need to reevaluate that constraint's violations. If the neighbourhood being searched can only affect a single constraint then the amount of time saved could lead to significant performance gains.

References

1. Mladenović, N., Hansen, P.: Variable neighborhood search. Computers and Operations Research **24**(11) (November 1997) 1097–1000
2. Viana, A., de Sousa, J.P., Matos, M.A.: Constraint oriented neighbourhoods - a new search strategy in metaheuristics. In Ibaraki, T., Nonobe, K., Yagiura, M., eds.: Metaheuristics: Progress as Real Problem Solvers. Volume 32 of Operations Research / Computer Science Interfaces Series. Springer (2005)
3. Beasley, J.E.: Or-library http://people.brunel.ac.uk/ mastjjb/jeb/info.html (October 2005)
4. Di Gaspero, L., Schaerf, A.: A composite-neighbourhood tabu search approach to the travelling tournament problem. Journal of Heuristics **13**(2) (January 2007) 189–207
5. Ågren, M., Flener, P., Pearson, J.: On constraint-oriented neighbours for local search. Technical Report 2007-009, Department of Information Technology, Uppsala University, Box 337, SE - 751 05 Uppsala, Sweden (March 2007)
6. Van Hentenryck, P., Michel, L.: Constraint-Based Local Search. The MIT Press (2005)