Fast Node Overlap Removal—Correction

Tim Dwyer[1], Kim Marriott[1], and Peter J. Stuckey[2]

[1] Clayton School of IT, Monash University, Australia
{tdwyer,marriott}@mail.csse.monash.edu.au
[2] NICTA Victoria Laboratory
Dept. of Comp. Science & Soft. Eng., University of Melbourne, Australia
pjs@cs.mu.oz.au

Keywords: graph layout, constrained optimization, separation constraints.

1 Introduction

Our recent paper [1] details an algorithm for removing overlap between rectangles, while attempting to displace the rectangles by as little as possible. The algorithm is primarily motivated by the node-overlap removal problem in graph drawing. The algorithm treats x- and y-dimensions separately, each as an instance of the *variable placement with separation constraints (VPSC)* problem:

> Given n variables $v_i \in V$, a weight $w_i \geq 0$ and a desired value d_i for each variable and a set of separation constraints C over these variables find an assignment to the variables which minimizes $\sum_{i=1}^{n} w_i \times (v_i - d_i)^2$ subject to C.

Each separation constraint $c \in C$ has form $u+g \leq v$ where $g \geq 0$ is the minimum separation between variables u and v.

In [1] we gave a procedure, *satisfy_VPSC*, that was intended to find a feasible but possibly non-optimal solution to the VPSC. Unfortunately, the algorithm we gave for *satisfy_VPSC* contained an error which means that in rarely occurring cases it may return an infeasible solution.

The basic approach of the algorithm *satisfy_VPSC* was to merge variables into larger and larger "blocks" of variables connected by active constraints. The algorithm processed the variables from smallest to greatest based on some total order given by the relation $\preceq_C$ where $u \preceq_C v$ iff there is a constraint $c \in C$ of form $u + g \leq v$.

Naive implementation of this algorithm has worst-case complexity of $O(|V| \cdot |C|)$. In order to improve efficiency, the algorithm given in [1] used a priority queue for each block b to store the block's "in" constraints, i.e. those constraints of form $u + g \leq v$ where v is in block b, ordered by their violation. When two blocks were merged so were their priority queues. Implementing the priority queues as *pairing heaps* [2] improved the amortized worst case complexity of the algorithm to $O((|V| + |C|) \log |C|)$.

Unfortunately, we have subsequently realized that in rare cases merging of priority queues meant that the algorithm given in [1] could return a infeasible solution. The problem is that when a block b is moved the violation of its "in" constraints changes value. It was claimed in [1], that the relative ordering of the constraints in the priority queue will not be changed as the result of a block's movement since all variables in the block will be moved by the same amount and so the violation of the constraints will be changed by the same amount for all non-internal constraints. This is true for the constraints in the priority queue of the current active block b but may not be true for constraints in a priority queue of a non-active block b' whose constraints refer to variables that are in b. This may mean that the priority queue of b' becomes invalid and, if at some future time b' becomes merged with the active block, can lead to an error.

It is relatively straightforward to fix the problem. We keep a time stamp for each block b, which is the time it last moved, and a time stamp for each constraint c in a priority queue implemented as a pairing heap, which is the time it was placed in the priority queue. When a constraint c is encountered in the priority queue of the currently active block b (as the result of coming to the "top" of the heap during a remove operation) we check that the other block b' that it refers to has not been moved since c was placed in the priority queue. If this is not true, the relative placement of the constraint in the queue could be wrong, so a down heap operation is performed on the constraint. A down heap is required since c's violation relative to the other constraints in the priority queue can only have decreased as block b' must have moved to the left. Since c's violation has decreased it is quite safe to lazily correct the problem.

The worst case complexity of the the corrected *satisfy_VPSC* is now $O((|V| \cdot |C|) \log |C|)$ since we might perform a down-heap operation repeatedly on the same constraint. However, if a depth-first traversal is used to construct the total ordering from the partial order, it is quite rare for this scenario to arise and in this case we believe that the expected amortized time complexity is $O((|V| + |C|) \log |C|)$. The revised algorithm is described in more detail in [3].

References

1. Dwyer, T., Marriott, K., Stuckey, P.: Fast node overlap removal. In: Proceedings of the 13th International Symposium on Graph Drawing (GD'05). Volume 3843 of LNCS. (2006) 153–164
2. Weiss, M.A.: Data Structures and Algorithm Analysis in Java. Addison Wesley Longman (1999)
3. Dwyer, T., Marriott, K., Stuckey, P.: Fast node overlap removal correction. Available from: www.csse.monash.edu.au/~tdwyer/FastNodeOverlapRemovalCorrection.pdf (2006)