Fast Node Overlap Removal — Addendum

Technical Report*, January 2006

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

[1] School of Comp. Science & Soft. Eng., 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

Abstract. This document highlights an oversight in our recent paper on a method for node overlap removal [1, 2]. The error, based on an incompleted specified invariant, occurs in the algorithm *satisfy_VPSC* and leads to a rarely occurring case where not all constraints are satisfied. We give the required additions to the algorithm to obtain correct behaviour, revise the worst case complexity theorem and reproduce the experimental performance data. While the worst case complexity is $O(n^2)$ we show that for typical input the performance is $O(n \log n)$ and this is reflected by the new experimental results.

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. That is, many graph drawing algorithms treat nodes as points with zero width and height so that, after a layout is found, if the nodes have labels or associated graphics the layout must be adjusted to remove any overlaps. The algorithm treats x- and y-dimensions separately, each as an instance of the *variable placement with separation constraints (VPSC)* problem as detailed below. The method for solving the VPSC problem to optimality is described in two parts. The *Satisfy_VPSC* procedure finds a solution in which all overlap is removed, but which may not necessarily be optimal. The *Solve_VPSC* algorithm uses *Satisfy_VPSC* to find an initial feasible solution, and then refines the arrangement until an optimal solution is found. The problem described and corrected in this paper occurs in *Satisfy_VPSC* where the algorithm, as originally described, could potentially produce infeasible solutions.

The problem stems from an erroneous assumption that, since variables were being processed left to right (while solving in the x-dimension) in a partial order determined by the directed acyclic graph of separation constraints, once a variable was placed no other variables upon which its constraints were dependant would be moved again. The algorithm relied on this assumed invariant to maintain heap data structures of incoming constraints to blocks of variables. That is, the heaps required that the order of relative slackness of incoming constraints to each block be preserved so that the topmost constraint on each heap would always be the most violated. The actual situation turns out to be slightly more complicated. The revised invariant, upon which the modified algorithm depends, is stated and proven below. The complete, correct *satisfy_VPSC* algorithm is also given and the statement of complexity modified. Finally, we compare experimental results for the new version of the algorithm with those from the original paper and find that practical performance is not adversely affected.

2 The *Satisfy_VPSC* Algorithm

In each pass of the node-overlap removal process we must solve the following constrained optimization problem for each dimension:

> *Variable placement with separation constraints (VPSC) problem.* Given n variables $v_1, \ldots, v_n$, a weight $v_i.weight \geq 0$ and a desired value $v_i.des$[1] for each variable and a set of separation constraints C over these variables find an assignment to the variables which minimizes $\sum_{i=1}^{n} v_i.weight \times (v_i - v_i.des)^2$ subject to C.

We can treat a set of separation constraints C over variables V as a weighted directed graph with a node for each $v \in V$ and an edge for each $c \in C$ from $left(c)$ to $right(c)$ with weight $gap(c)$. We call this the *constraint graph*. We define $out(v) = \{c \in C \mid left(c) = v\}$ and $in(v) = \{c \in C \mid right(c) = v\}$. Note that edges in this graph are *not* the edges in the original graph.

We restrict attention to VPSC problems in which the constraint graph is acyclic and for which there is at most one edge between any pair of variables. It is possible to transform an arbitrary satisfiable VPSC problem into a problem of this form and our generation algorithm will generate constraints with this property.

Since the constraint graph is acyclic it imposes a partial order on the variables: we define $u \preceq_C v$ iff there is a (directed) path from u to v using the edges in separation constraint set C. We will make use of the function *total_order*(V,C) which returns a total ordering for the variables in V, i.e. it returns a list $[v_1, \ldots, v_n]$ s.t. for all $j > i$, $v_j \not\preceq_C v_i$.

Figure 1 lists the basic algorithm for finding a solution to the VPSC problem such that the separation constraints are satisfied and the variable placement is "close" to optimal. It takes as input a set of separation constraints C and a set of variables V. The algorithm works by merging variables into larger and larger "blocks" of contiguous variables connected by a spanning tree of active constraints, where a separation constraint $u + a \leq v$ is active if, for the current position for u and v, $u + a = v$.

[1] $v_i.des$ is set to x^0_{vi} or y^0_{vi} for each dimension, as used in $generate\_C^{no}_{\{x|y\}}$.

```
procedure satisfy_VPSC(V,C)
    timeCtr ← 0
    [v_1, ..., v_n] ← total_order(V, C)
    for i ∈ 1...n do
        merge_left(block(v_i))
    endfor
return [v_1 ← posn(v_1), ..., v_n ← posn(v_n)]

procedure block(v)
    let b be a new block s.t.
        b.vars ← {v}
        b.nvars ← 1
        b.posn ← v.des
        b.wposn ← v.weight × v.des
        b.weight ← v.weight
        b.active ← ∅
        b.in ← newQueue()
        b.time ← timeCtr ← timeCtr + 1
        for c ∈ in(v) do
            time(c) ← timeCtr
            add(b.in, c)
        endfor
    block[v] ← b
    offset[v] ← 0
return b

procedure merge_left(b)
    while violation(top(b.in)) > 0 do
        c ← top(b.in)
        removeTop(b.in)
        bl ← block[left(c)]
        if bl.in =null then
            setup_in_constraints(bl)
        endif
        distbltob ← offset[left(c)] + gap(c)
                    −offset[right(c)]
        if b.nvars > bl.nvars then
            merge_block(b, c, bl, −distbltob)
        else
            merge_block(bl, c, b, distbltob)
            b ← bl
        endif
    endwhile
return

procedure merge_block(p, c, b, distptob)
    p.wposn ← p.wposn + b.wposn−
                distptob × b.weight
    p.weight ← p.weight + b.weight
    p.posn ← p.wposn/p.weight
    p.active ← p.active ∪ b.active ∪ {c}
    for v ∈ b.vars do
        block[v] ← p
        offset[v] ← distptob + offset[v]
    endfor
    p.vars ← p.vars ∪ b.vars
    p.nvars ← p.nvars + b.nvars
    timeCtr ← timeCtr + 1
    top(p.in)
    top(b.in)
    p.in ← merge(p.in, b.in)
    b.time ← timeCtr
return
```

Fig. 1. Algorithm *satisfy_VPSC*(V, C) to satisfy the Variable Placement with Separation Constraints (VPSC) problem

```
procedure greater_than(c,d)
    v_c ← violation(c)
    if block[left(c)].time > time(c)
        or block[left(c)] = block[right(c)]
    then
        v_c ← ∞
    endif
    v_d ← violation(d)
    if block[left(d)].time > time(d)
        or block[left(d)] = block[right(d)]
    then
        v_d ← ∞
    endif
return v_c > v_d
```

```
procedure top(heap)
    outOfDate ← ∅
    while not empty(heap) do
        c ← heap.root
        l ← block[left(c)]
        r ← block[right(c)]
        if l = r then
            removeTop(heap)
        else if l.time > time(c) then
            removeTop(heap)
            outOfDate ← outOfDate ∪ {c}
        else
            break
        endif
    endwhile
    for c ∈ outOfDate do
        time(c) ← timeCtr
        insert(heap, c)
    endfor
return heap.root
```

Fig. 2. New procedures for handling the constraint pairing heaps.

We represent a block b using a record with the following fields: *vars*, the set of variables in the block; *nvars*, the number of variables in the block; *active*, the set of constraints between variables in the block which form the spanning tree of active constraints; *in*, which (essentially) contains the set of constraints $\{c \in C \mid right(c) \in b.vars \text{ and } left(c) \notin b.vars\}$; *out*, the set of out-going constraints defined symmetrically to *in*; *posn*, the position of the block's "reference point"; *wposn*, the sum of the weighted desired locations of variables in the block; and *weight*, the sum of the weights of the variables in the block. In this new version of the algorithm we have added the field *time* which indicates when the set *in* was last examined or modified.

In addition, the algorithm uses two arrays *blocks* and *offset* indexed by variables where $block[v]$ gives the block of variable v and $offset[v]$ gives the distance from v to its block's reference point. Using these we define the function $posn(v) = block[v].posn + offset[v]$ which gives the current position of variable v.

The constraints in the field $b.in$ for each block b are stored in a priority queue such that the function $top(q)$ (see Figure 2) always returns the most violated contraint in the queue q where $violation(c) = posn(left(c)) + gap(c) - posn(right(c))$. We explain the implementation of these queues below.

The main procedure, *satisfy_VPSC*, processes the variables based on a total order induced from a topological sort of the constraint graph. At each stage the invariant is that we have found an assignment to $v_1, .., v_{i-1}$ which satisfies the separation constraints. We process vertex v_i as follows. First we assign v_i to its own block, created using the function *block* and we place this block at $v_i.des$. Of course the problem is that some of the "in" constraints may be violated. We check for this and find the most violated constraint c. We then merge the two blocks connected by c using the function *merge_block*. This merges the two blocks into a new block with c as the active connecting constraint. We repeat this until the block no longer overlaps the preceding block, in which case we have found a solution to $v_1, .., v_i$.

At each step we place the reference point $b.posn$ for each block at its optimum position, i.e. the weighted average of the desired positions:

$$\frac{\sum_{i=1}^{k} v_i.weight \times (\mathit{offset}[v_i] - v_i.des)}{\sum_{i=1}^{k} v_i.weight}$$

In order to efficiently compute the weighted arithmetic mean when merging two blocks we use the fields *wposn*, the sum of the weighted desired locations of variables in the block and *weight* the sum of the weights of the variables in the block.

We use four queue functions: $newQueue()$ which returns a new queue, $add(q, c)$ which inserts the constraint c into the queue q, $top(q)$ which returns the constraint in q with maximal violation, $remove(q)$ which deletes the top constraint from q, and $merge(q_1, q_2)$ which returns the queue resulting from merging queues q_1 and q_2. There are two special conditions that our queues must handle. The first is that some of the constraints in $b.in$ may be *internal* constraints, i.e. constraints which are between variables in the same block. Such internal constraints are removed from the queue when encountered by $top(q)$. The other condition is that when a block is moved, *violation* for each of the incoming and outgoing constraints changes value. Therefore to avoid a complete scan of all incoming constraints to find the most violated we take advantage of how blocks move relative to each other to maintain lazily updated priority queues based on *pairing heaps* [3] with efficient support for the above operations. The operation of these queues is dependant on the following conditions.

Lemma 1. *Let $u + d \leq v$ be a constraint over variables u and v. Let $a = block[u], b = block[v]$ and let the constraint between u and v be the most violated constraint in $b.in$. Then, for any $w \in b.vars$, if $p_w = posn(w)$ before the merge and $p'_w = posn(w)$ after the merge, then $p'_w > p_w$. Symmetrically, for any $m \in a.vars$, $p'_m < p_m$*

Proof. All variables in $a.vars$ and $b.vars$ are offset by a fixed amount from their reference positions $a.posn$ and $b.posn$ respectively. We can therefore W.O.L.G. rewrite the constraint as $a.posn + d \leq b.posn$. In the *merge_block* procedure we obtain a new position p for the merged block as the weighted average position s.t. $p \cdot (a.weight + b.weight) = a.weight \cdot a.posn + b.weight \cdot (b.posn - d)$. For the constraint to be violated before the merge we must have that $b.posn - d < a.posn$. Combining either side of this inequality with the expression above we are able to eliminate the sum of weights and find that $p > b.posn - d$ and $p < a.posn$. Thus, variables in the block at the RHS of the constraint must increase in value and those on the LHS will decrease. □

Lemma 2. *Given the call merge_left(block(v)) (i.e. the first call to merge_left for block[v]) for some variable $v \in V$ with position $p_v = posn(v)$ prior to the call and subsequent position p'_v, $p'_v \geq p_v$. Conversly, for any variable $u \in V, u \neq v$ with position p_u prior to merge_left(block(v)) and subsequent position p'_u, $p'_u \leq p_u$.*

Proof. Since *merge_left* only corrects violated constraints incoming to the argument block, $p'_v > p_v$ by Lemma 1 if such constraints exist or $p'_v = p_v$ otherwise. Again, since we only merge across incoming constraints, any u where $u \not\preceq_C v$ or $u \preceq_C v$ where there is unsufficient violation in the constraints in the path from u to v for $block[u]$ and $block[v]$ to be merged, will be unaffected by the call $merge\_left(block(v))$, so $p'_u = p_u$. If $u \preceq_C v$ and constraints in this path are violated, then when $block[u]$ is first merged with $block[v]$ it will be across a constraint incoming to $block[v]$ and so by Lemma 1 $posn(u)$ must decrease. Such a decrease in position for the variables in $block[u]$ may lead to further violations which must be corrected by some increase in position as *merge_left* recurses, however since all such constraints were satisfied prior to the initial call, subsequent increases must be smaller and hence the net effect is $p'_u < p_u$. □

Theorem 1. *Let c be the constraint at the top of the heap for block r with LHS in block $l \neq r$. If $time[c] > l.time$ then c is the most violated incoming constraint of block r.*

Proof. (Sketch) The max-heap condition that is (lazily) maintained by the pairing heaps used in incoming constraint priority queues for each block is:

> For any two constraints c, d in a particular heap, if c is positioned as the parent of d then $greater\_than(c, d)$ is true.

Block time stamps are updated when blocks are created or when a block is merged. Thus, for some variable v any time $posn(v)$ can change, $block[v].time$ is updated. Constraint time stamps are updated whenever constraints are placed in a queue. When a block b on the right side of a constraint moves, the *violation* of each constraint in $b.in$ is changed by the same amount and the relative order of constraints in the queue is not affected. However, a change in position of the LHS of the constraint can happen independently to constraints incoming to the RHS. But since we apply *merge_left* to variables in an order due to a topological-sort over the constraint DAG such a movement cannot be due to an initial call to *merge_left* and therefore, by Lemma 2 must be a decrease in position and hence a decrease in the constraint's degree of violation relative to the other constraints in the RHS queue. Thus, it may be higher in the heap than it should be. However, since it is decreased, if its parent satisfied the max-heap condition before the change, that parent must still satisfy this condition. That is, the parent must still be more violated than any of its children. The check $time[c] > l.time$ tells us that the LHS of c has not been moved since c was placed in the queue, that the max-heap condition of c relative to its children must hold, and therefore if c is the root of the heap, then it must be the most violated constraint in the entire heap. □

Thus, the procedure *top* in Figure 2 is able to obtain the most violated constraint in the priority queue by removing constraints that fail the timestamp test until a valid one is found. The out-of-date constraints are then reinserted into the heap with an updated timestamp. After all are reinserted, the root of the heap is returned as the most violated.

The last detail concerns the merging of constraint queues in the *merge_block* operation, see Figure 1. A pairing-heap merge operation simply compares the roots of the two heaps, takes the maximum as the new root, and makes the other heap a child of this root. To ensure our invariant holds after a merge we first apply the *top* operation to each heap so that the roots are correct.

Theorem 2. *Let θ be the assignment to the variables V returned by* satisfy_VPSC(V, C). *Then θ satisfies the separation constraints C.*

Proof. (Sketch) The induction hypothesis is that after processing variable v_i we have found a solution θ_i to the variables $V_i = \{v_1, \ldots, v_i\}$ which satisfies the constraints $C_i = \{c \in C \mid \{end(c), in(c)\} \subseteq V_i\}$.

Clearly this holds for the base case when $i = 0$.

Now consider v_{i+1}. We will now iteratively construct the block b containing this variable. At each step we have the following invariant that the only constraints in C_{i+1} that may not hold are non-internal constraints in $b.in$, i.e.

$$\{c \in C_{i+1} | in(c) \in b.vars \wedge out(c) \notin b.vars\}.$$

Furthermore, we have that for all $v \in V_i$ $posn(v) = \theta_i(v)$ if $v \notin b.vars$ or if $v \in b.vars$, $posn(v) \leq \theta_i(v)$

Clearly these hold when b contains only the variable v_{i+1} since because of the total ordering $C_{i+1} \setminus C_i = in(v_{i+1})$.

Now consider a "merge" step in which the most violated non-internal constraint $c \in b.in$ has been selected and bl is the block of $left(c)$. Let b' be the block resulting from merging b and bl. Since the merge moves variables in b and bl uniformly no internal constraint in either b or bl can become unsatisfied. Furthermore since c is the most violated constraint between b and bl no other constraint between the two can be violated once b and bl have been merged. Since we place the variables at the weighted average of the desired values of the variables we have that $v \in b.vars$, $posn(v) \leq \theta_i(v)$. Thus since $posn(v) = \theta_i(v)$ if $v \notin b.vars$, the only possibly violated constraints are non-internal constraints in b'.

Theorem 3. *The procedure* satisfy_VPSC(V, C) *has amortized complexity* $O((|V| + |C|) \log |C|)$.

Proof. (Sketch) Computing the initial total order over the directed acyclic graph of constraints takes $O(|V| + |C|)$ time with depth first search.

Pairing-heaps give amortized $O(1)$ *insert*, *findMin* (*top*) and *merge* operations while *remove* is $O(\log m)$ (amortized) in m the size of the heap. Since internal constraints may be merged into the heaps we may perform at most

m remove operations in eliminating them. Thus, maintenance of *in* and *out* constraint queues in satisfy_VPSC is $O(m \log m)$. Since each constraint cannot appear more than once in the priority queues and since we do not reinsert any constraints after removing them, we have $m \leq |C|$.

The other potentially costly part of merging is copying the contents of blocks. We perform at most $n \leq \min(|C|, |V| - 1)$ merges since we can only merge as many times as there are constraints and after $|V| - 1$ merges we are left with a single block. Since we always copy the smaller block into the larger each variable is copied up to $\log n$ times, the worst case occurring when merging equally sized blocks for each merge — proof is by a standard recurrence relation. Thus, the total cost of copying variables is $|V| \log n$.

From the bounds on n and m we have that the outer-most *for* loop in *satisfy_VPSC* is within $O((|C| + |V|) \log |C|)$ time which also subsumes the initial cost of computing the total order.

3 Results

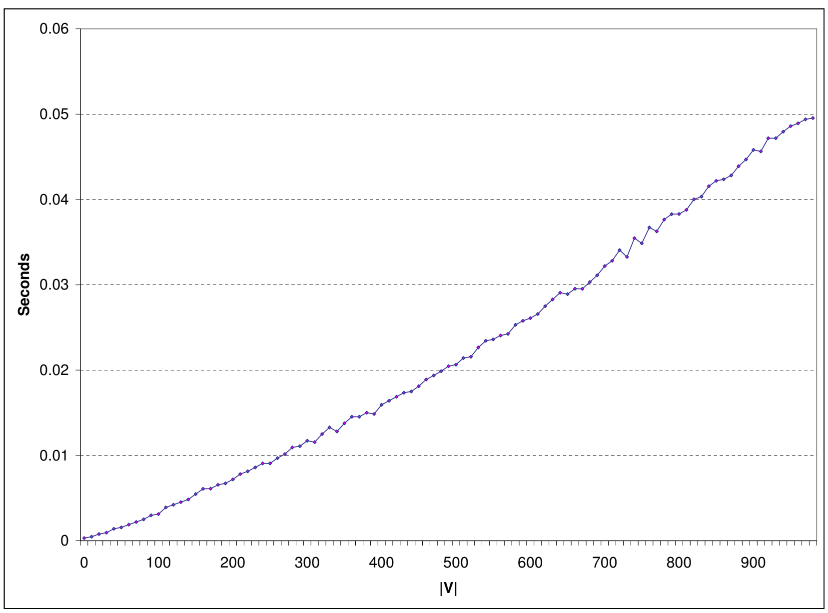

Fig. 3. Running times for overlap removal with *satisfy_vpsc*.

Figure 3 gives running time results for overlap removal with *satisfy_vpsc* applied to sets of randomly generated rectangles. The time includes constraint generation time and three passes of *satisfy_vpsc* — applied horizontally, then vertically, then horizontally once more. We varied the number of rectangles between 10 and 1000, generated but adjusted the size of the rectangles to keep k (the average number of overlaps per rectangle) appoximately constant ($k \approx 10$). Each size sample was run 100 times and the time shown at each point is the mean.

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. Dwyer, T., Marriott, K., Stuckey, P.: Fast node overlap removal. Technical Report 2005/175, Monash University, School of Computer Science and Software Engineering (2005)
3. Weiss, M.A.: Data Structures and Algorithm Analysis in Java. Addison Wesley Longman (1999)