Fast Node Overlap Removal in Graph Layout Adjustment

Technical Report*, August 2005

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. Most graph layout algorithms in the field of graph drawing treat nodes as points. The problem of node overlap removal is to adjust the layout generated by such methods so that nodes of non-zero width and height do not overlap, yet are as close as possible to their original positions. We give an $O(n \log n)$ algorithm for achieving this assuming that the number of nodes overlapping any single node is bounded by some constant. This method has two parts, a constraint generation algorithm which generates a linear number of "separation" constraints and an algorithm for finding a solution to these constraints "close" to the original node placement values. We also extend our constraint solving algorithm to give an active set based algorithm which is guaranteed to find the optimal solution but which has considerably worse theoretical complexity. We compare our method with convex quadratic optimization and force scan approaches and find that it is faster than either, gives results of better quality than force scan methods and similar quality to the quadratic optimisation approach.

Keywords: graph layout, constrained optimization, separation constraints

1 Introduction

Graph drawing has been extensively studied over the last twenty years [1]. However, most research has dealt with *abstract graph layout* in which nodes are treated as points. Unfortunately, this layout strategy is inadequate in many applications since nodes frequently have labels or icons and a layout for the abstract graph may lead to overlapping nodes when these are added. While a few attempts have been made at designing layout algorithms that consider node size

* An abridged version of this paper is to appear in the proceedings of the 13^{th} international symposium on Graph Drawing, Lecture Notes in Computer Science, Springer, 2005

(e.g. [2–4]), the approaches are specific to certain layout styles and to the best of the authors' knowledge none are perfect in all situations.

For this reason, a number of papers, e.g. [5–10], have described algorithms for performing *layout adjustment* in which an initial graph layout is modified so that node overlapping is removed. The underlying assumption is that the initial graph layout is good so that this layout should be preserved when removing the node overlap. Lyons et al.[10] offered a technique based on iteratively moving nodes to the centre of their Voronoi cells until crossings are removed. Misue et al. [5] propose several models for a user's "mental map" based on *orthogonal ordering*, *proximity relations* and *topology* and define a simple heuristic *Force Scan* algorithm (FSA) for node-overlap removal that preserves orthogonal ordering. Hayashi et al. [7] propose a variant algorithm (FSA′) that produces more compact drawings while still preserving orthogonal ordering. They also show that this problem is NP-complete. Various other improvements to the FSA method exist and a survey is presented by Li et al. [11]. More recently, Marriott et al. [6] investigated a quadratic programming (QP) approach which minimises displacement of nodes while satisfying non-overlap constraints. Their results demonstrate that the technique offers results that are preferable to FSA in a number of respects, but require significantly more processing time. In this paper we address the last issue.

Our contribution consists of two parts: first we detail a new algorithm for computing the linear constraints to ensure non-overlap in a single dimension. This has worst case complexity $O(n \log n)$ where n is the number of nodes and generates $O(n)$ non-overlap constraints.[1] Previous approaches have had quadratic or cubic complexity and as far as we are aware it has not been previously realized that only a linear number of non-overlap constraints are required.

Each non-overlap constraint has the form $u + a \leq v$ where u and v are variables and $a \geq 0$ is a constant. Such constraints are called separation constraints. Our second contribution is to give a simple algorithm for solving quadratic programming problems of the form: minimize $\sum_{i=1} v_i.weight \times (v_i - v_i.des)^2$ subject to a conjunction of separation constraints over variables $v_1, \ldots, v_n$ where $v_i.des$ is the desired value of variable v_i and $v_i.weight \geq 0$ the relative importance.

We give two versions of the algorithm. The first version has $O(n + m \log m)$ worst case complexity where m is the number of constraints and n the number of variables. It is not guaranteed to find an optimal solution but in practice it works very well. The second version of the algorithm is guaranteed to find an optimal solution but its worst case complexity may be exponential. However in practice it is reasonably fast. Importantly these algorithms do not require the use of a complex mathematical programming software.

Together these two algorithms give us an $O(n \log n)$ algorithm to remove overlap between n nodes. We provide an empirical evaluation of our approach and compare it to the original QP approach and to FSA′ of Hayashi et al. [7] (henceforth we refer to this simply as FSA) and the Voronoi approach of [10].

[1] Assuming that the number of nodes overlapping a single node is bounded by some constant k.

We find that it is considerably faster than the original QP approach and in practice has speed better than FSA. However, it still produces layout of quality comparable to the QP approach and considerably better than that of FSA.

2 Background

We assume that we are given a graph G with nodes $V = \{1, \ldots, n\}$, a width, w_v, and height, h_v, for each node $v \in V$,[2] and an initial layout for the graph G, in which each node $v \in V$ is placed at (x_v^0, y_v^0). We assume that no two nodes are placed at exactly the same initial position (unlikely given a sensible layout). If this is not the case we perturb one position slightly.

We are concerned with layout adjustment: we wish to preserve the initial graph layout as much as possible while removing all node label overlapping. A natural heuristic to use for preserving the initial layout is to require that nodes are moved as little as possible. This corresponds to the Proximity Relations mental map model of Misue et al. [5].

Following [6] we define the *layout adjustment problem* to be the constrained optimization problem: minimize ϕ_{change} subject to C^{no} where the variables of the layout adjustment problem are the x and y coordinates of each node $v \in V$, x_v and y_v respectively, and the objective function minimizes node movement $\phi_{change} = \phi_x + \phi_y = \sum_{v \in V}(x_v - x_v^0)^2 + (y_v - y_v^0)^2$, and the constraints C^{no} ensure that there is no node overlapping. That is, for all $u, v \in V$, $u \neq v$ implies

$$\begin{array}{l} x_v - x_u \geq \frac{1}{2}(w_v + w_u) \text{ } (v \text{ right of } u) \vee x_u - x_v \geq \frac{1}{2}(w_v + w_u) \text{ } (u \text{ right of } v) \\ \vee\, y_v - y_u \geq \frac{1}{2}(h_v + h_u) \text{ } (v \text{ above } v) \quad \vee\, y_u - y_v \geq \frac{1}{2}(h_v + h_u) \text{ } (u \text{ above } v) \end{array}$$

A variant of this problem is when we additionally require that the new layout preserves the *orthogonal ordering* of nodes in the original graph, i.e., their relative ordering in the x and y directions. This is a heuristic to preserve more of the original graph's structure. Define $C_x^{oo} = \bigwedge\{x_v \geq x_u \mid x_v^0 \geq x_u^0\}$ and C_y^{oo} equivalently for y. The orthogonal ordering problem adds $C_x^{oo} \wedge C_y^{oo}$ to the constraints to solve.

Our approach to solving the layout adjustment problem is based on [6] where quadratic programming is used to solve a linear approximation of the layout adjustment problem. The basic algorithm is given in Figure 1.

There are two main ideas behind the quadratic programming approach. The first is to approximate each non-overlap constraint in C^{no} by one of its disjuncts. The second is to split it into two separate optimization problems, one for the x dimension and one for the y dimension, by breaking the optimization function into two parts and the constraint into two parts. Separating the problem in this way improves efficiency by reducing the number of constraints considered in each problem and if, say, we solve for the x direction first, it allows us to delay the computation of C_y^{no} to take into account the node overlapping which has been

[2] Any extra padding required to ensure a minimal separation between nodes is included in w_v and h_v

quadratic-opt
  compute C_x^{no}
  $\boldsymbol{x} :=$ minimize ϕ_x subject to C_x^{no}
  $\boldsymbol{x}^0 := \boldsymbol{x}$
  compute C_y^{no}
  $\boldsymbol{y} :=$ minimize ϕ_y subject to C_y^{no}

Fig. 1. Quadratic programming approach to layout adjustment

removed by the optimization in the x direction. Thus separation allows us to find a better solution.

3 Generating Non-Overlap Constraints

It is relatively simple to generate the non-overlap constraints in each dimension in $O(|V| \cdot \log |V|)$ time using a line-sweep algorithm related to standard rectangle overlap detection methods [12]. First consider the generation of horizontal constraints. We use a vertical sweep through the nodes, keeping a horizontal "scan line" list of open nodes with each node having references to its closest left and right neighbors (or more exactly the neighbors with which it is currently necessary to generate a non-overlap constraint). When the scan line reaches the top of a new node, this is added to the list and its neighbors computed. When the bottom of a node is reached the the separation constraints for the node are generated and the node is removed from the list.

The detailed algorithm is shown on the left of Figure 2. It uses a vertically sorted list of events to guide the movement of the scan line *scan_line*. An event is a record with three fields, *kind* which is either *open* or *close* respectively indicating whether the top or bottom of the node has been reached, *node* which is the node name, and *posn* which is the vertical position at which this happens, i.e. the top or bottom of the node.

The *scan_line* stores the currently open nodes. We use a red-black tree to provide $O(\log |V|)$ *insert*, *remove*, *next_left* and *next_right* operations. An empty scan line is constructed with *new* and the functions *insert* and *remove* respectively add and remove a node from the scan line, returning the resulting scan line. The functions $next\_left(scan\_line, v)$ and $next\_right(scan\_line, v)$ return the closest neighbor to the left and, respectively, the right of node v in the scan line.

The functions $get\_left\_nbours(scan\_line, v)$ and $get\_right\_nbours(scan\_line, v)$ respectively detect the neighbours to the left and the right with which node v should have non-overlap constraints. These are heuristics. It seems reasonable to set up a non-overlap constraint with the closest non-overlapping node on each side and a subset of the overlapping nodes. One choice for *get_left_nbours* is shown in Figure 2. This makes use of the functions

$$olap_x(u, v) = (w_u + w_v)/2 - |x_u^0 - x_v^0|$$
$$olap_y(u, v) = (h_u + h_v)/2 - |y_u^0 - y_v^0|$$

```
procedure generate_C_x^no(V)
events := { event(open, v, y_v − h_v/2),
            event(close, v, y_v + h_v/2) | v ∈ V}
[e_1, ..., e_2n] := events sorted by posn
scan_line := new()
for each e_1, ..., e_2n do
    v := e_i.node
    if e_i.kind = open then
        scan_line := insert(scan_line, v)
        leftv := get_left_nbours(scan_line, v)
        rightv := get_right_nbours(scan_line, v)
        left[v] := leftv
        for each u ∈ leftv do
            right[u] := (right[u] ∪ {v}) \ rightv
        right[v] := rightv
        for each u ∈ rightv do
            left[u] := (left[u] ∪ {v}) \ leftv
    else /* e_i.kind = close */
        for each u ∈ left[v] do
            generate x_u + (w_u + w_v)/2 ≤ x_v
            right[u] := right[u] \ {v}
        for each u ∈ right[v] do
            generate x_v + (w_u + w_v)/2 ≤ x_u
            left[u] := left[u] \ {v}
        scan_line := remove(scan_line, v)
return

function get_left_nbours(scan_line, v)
u := next_left(scan_line, v)
while u ≠ NULL do
    if olap_x(u, v) ≤ 0 then
        leftv := leftv ∪ {u}
        return leftv
    if olap_x(u, v) ≤ olap_y(u, v) then
        leftv := leftv ∪ {u}
    u := next_left(scan_line, u)
return leftv
```

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

procedure merge_left(b)
while violation(top(b.in)) > 0 do
    c := top(b.in)
    b.in := remove(c)
    bl := block[left(c)]
    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
return

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 := add(new(), in(v))
block[v] := b
offset[v] := 0
return b

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]
p.in := merge(p.in, b.in)
p.vars := p.vars ∪ b.vars
p.nvars := p.nvars + b.nvars
return
```

Fig. 2. Algorithm *generate*_$C_x^{no}(V)$ to generate horizontal non-overlap constraints between nodes in V, and algorithm *satisfy_VPSC*(V, C) to satisfy the Variable Placement with Separation Constraints (VPSC) problem

which respectively measure the horizontal and vertical overlap between nodes u and v. The main loop iteratively searches left until the first non-overlapping node to the left is found or else there are no more nodes. Each overlapping node u found on the way is collected in $leftv$ if the horizontal overlap between u and v is less than the vertical overlap.

The arrays $left$ and $right$ respectively detail for each open node v the nodes to the left and to the right for which non-overlap constraints should be generated. These are appropriately updated whenever a new node v is added. The only subtlety is that redundant constraints are removed, i.e. if there is currently a non-overlap constraint between any $u \in leftv$ and $u' \in rightv$ then it can be removed since it will be implied by the two new non-overlap constraints between u and v and v and u'.

Theorem 1. *The procedure generate_C_x^{no}(V) has worst-case complexity $O(|V| \cdot k(\log |V| + k)$ where k is the maximum number of nodes overlapping a single node with appropriate choice of heap data structure. Furthermore, it will generate $O(k \cdot |V|)$ constraints.*

Proof. See Appendix B

Assuming k is bounded, the worst case complexity is $O(|V| \log |V|)$.

Theorem 2. *The procedure generate_C_x^{no}(V) generates separation constraints C that ensure that if two nodes do not overlap horizontally in the initial layout then they will not overlap in any solution to C.*

Proof. Follows from the construction.

The code for *generate_*C_y^{no}, the procedure to generate vertical non-overlap constraints is essentially dual to that of *generate_*C_x^{no}. The only difference is that any remaining overlap must be removed vertically. This means that we need only find and return the single closest node in the analogue of the functions *get_left_nbours* and *get_right_nbours* since any other nodes in the scan line will be constrained to be above or below these. This means that the number of left and right neighbours is always 1 or less and gives us the following complexity results.

Theorem 3. *The procedure generate_C_y^{no}(V) has worst-case complexity $O(|V| \cdot \log |V|)$. Furthermore, it will generate no more than $2 \cdot |V|$ constraints.*

Theorem 4. *The procedure generate_C_y^{no}(V) generates separation constraints C that ensure that no nodes will overlap in any solution to C.*

4 Solving Separation Constraints

Non-overlap constraints c have the form $u + a \leq v$ where u, v are variables and $a \geq 0$ is the minimum gap between them. We use the notation $left(c)$, $right(c)$ and $gap(c)$ to refer to u, v and a respectively. Such constraints are called *separation constraints*. 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 $v_i.des$[3] 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 $j > i$, $v_j \not\preceq_C v_i$.

We first give a fast algorithm for finding a solution to the VPSC algorithm which satisfies the separation constraints and which is "close" to optimal. 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, $u + a = v$.

The generic algorithm is shown in the right of Figure 2. It takes as input a set of separation constraints C and a set of variables V.

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 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 top constraint in the queue is always the most violated

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

where $violation(c) = left(c) + gap(c) - right(c)$. We use four queue functions: $new()$ which returns a new queue, $add(q, C)$ which inserts the constraints in the set C into the queue q and returns the result, $top(q)$ 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. The only slight catch 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. The other slight catch is that when a block is moved *violation* changes value. However, the ordering induced by $violation(c)$ does not change since all variables in the block will be moved by the same amount and so $violation(c)$ will be changed by the same amount for all non-internal constraints. This consistent ordering allows us to implement the priority queues as *pairing heaps* [13] with efficient support for the above operations.

The main procedure, *satisfy_VPSC*, processes the variables from smallest to greatest based on a total order reflecting 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 placing it 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:

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

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.

Example 1. Consider the example of laying out the boxes A,B,C,D shown in Figure 3(a) each shown at their desired position 1.5, 3, 3.5, and 5 respectively and assuming the weights on the boxes are 1,1,2 and 2 respectively. The constraints generated by *generate_*C_x^{no} are $c_1 \equiv v_A + 2.5 \leq v_B$, $c_2 \equiv v_B + 2 \leq v_C$ and $c_3 \equiv v_B + 2 \leq v_D$. Assume the algorithm chooses the total order A,B,C,D. First we add block A, it is placed at its desired position as shown in Figure 3(a). Next we consider block B, $b.in = \{c_1\}$ and the violation of this constraint is 1. We retrieve *bl* as the block containing A. and calculate *distbltob* as 2.5. We now merge block B into the block containing A. The new block position is 1 as shown in Figure 3(b), and c_1 is added to the active constraints. Next we consider block C, we find it must merge with block AB. The new positions are shown in Figure 3(c). Since there is no violation with the block D, the final position leaves it where it is. The result is optimal

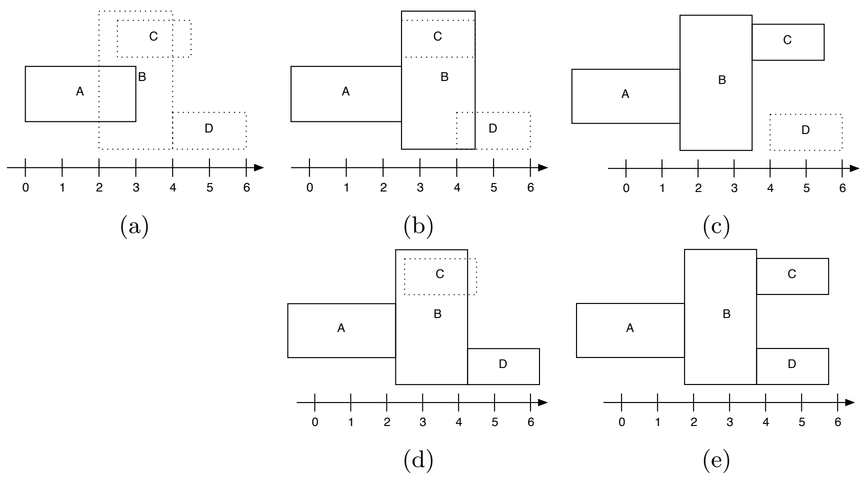


Fig. 3. Example of (non-optimal) algorithm for VPSC problem giving optimal (c) or non-optimal (e) answer

Theorem 5. *The assignment to the variables V returned by* satisfy_VPSC(V, C) *satisfies the separation constraints C.*

Proof. See Appendix B

Theorem 6. *The procedure* satisfy_VPSC(V, C) *has worst-case complexity* $O((|C|+|V|) \log |C|)$.

Proof. See Appendix B

Since each block is placed at its optimal position one might hope that the solution returned by *satisfy_VPSC* is also optimal. This was true for the example above. Unfortunately, as the following example shows it is not always true.

Example 2. Consider the same blocks as in Example 1 but with the total order A,B,D,C. The algorithm works identically to the stage shown in Figure 3(b). But now we consider block D, which overlaps with block AB. We merge the blocks to create block ABD which is placed at 0.75, as shown in Figure 3(d). Now block ABD overlaps with block C so we merge the two to the final position 0.166 as shown in Figure 3(e). The result is not optimal.

The solution will be non-optimal if we can improve the solution by splitting a block. This may happen if a merge becomes "invalidated" by a later merge. It is relatively straight-forward to check whether a solution is optimal by computing the Lagrange multipliers for the constraints. We must split a block at an active constraint c if its corresponding Lagrange multiplier λ_c is negative. Because of the simple nature of the separation constraints it is possible to compute λ_c (more exactly $\lambda_c/2$) for the active constraints in each block in linear time. We simply perform a depth-first traversal of the constraints in *b.active* summing $v.weight \times (posn(v) - v.des)$ for the variables below this variable in the tree. The algorithm is detailed in Figure 4. It assumes the data structures in *satisfy_VPSC* and stores $\lambda_c/2$ in the $lm[c]$ for each $c \in C$. For space reasons we leave the justification of this to the appendix.

```
procedure solve_VPSC(V,C)
satisfy_VPSC(V,C)
compute_lm()
while exists c ∈ C s.t. lm[c] < 0 do
    choose c ∈ C s.t. lm[c] < 0
    b := block[left(c)]
    lb := restrict_block(b, left(b, c))
    rb := restrict_block(b, right(b, c))
    rb.posn := b.posn
    rb.wposn := rb.posn × rb.weight
    merge_left(lb)
    /* original rb may have been merged */
    rb := block[right(c)]
    rb.wposn := Σ_{v∈rb} v.weight × (v.des − offset[v])
    rb.posn := rb.wposn/rb.weight
    merge_right(rb)
    compute_lm()
endwhile
return [v_1 ← posn(v_1), . . . , v_n ← posn(v_n)]
```

```
procedure compute_lm()
for each c ∈ C do lm[c] := 0 endfor
for each block b do
    choose v ∈ b.vars
    comp_dfdv(v, b.active, NULL)

function comp_dfdv(v, AC, u)
dfdv := v.weight × (posn(v) − v.des)
for each c ∈ AC s.t. v = left(c)
        and u ≠ right(c) do
    lm[c] := comp_dfdv(right(c), AC, v)
    dfdv := dfdv + lm[c]
for each c ∈ AC s.t. v = right(c)
        and u ≠ left(c) do
    lm[c] := − comp_dfdv(left(c), AC, v)
    dfdv := dfdv − lm[c]
return dfdv
```

Fig. 4. Algorithm to find an optimal solution to a VPSC problem with variables V and separation constraints C.

Using this it is relatively simple to extend *satisfy_VPSC* so that it computes an optimal solution. The algorithm is given in Figure 4. This uses *satisfy_VPSC* to find an initial solution to the separation constraints and calls *compute_lm* to compute the Lagrange multipliers. The main while loop checks if the current solution is optimal, i.e. if for all $c \in C$, $\lambda_c \geq 0$. If this is true the algorithm terminates since the optimal solution has been found. Otherwise one of the constraints $c \in C$ with a negative Lagrange multiplier is chosen (in our actual implementation we choose the constraint with the most negative multiplier) and the block b containing that constraint is split into two new blocks, lb which contains the variables in $left(b, c)$ and rb which contains those in $right(b, c)$. We define $left(b, c)$ to be the nodes in $b.vars$ connected by a path of constraints from $b.active \setminus \{c\}$ to $left(c)$, i.e. the variables which are in the left sub-block of b if b is split by removing c. Symmetrically, we define $right(b, c)$ to be the variables which are in the right sub-block of b if b is split by removing c. The split is done by calling the procedure *restrict_block*(b, V) which takes a block b and returns a new block restricted to the variables $V \subseteq b.vars$. For space reasons we do not include the (straight-forward) code for this.

Now the new blocks lb and rb are placed in their new positions using the procedures *merge_left* and *merge_right*. First we place lb. Since $lm[c] < 0$, lb wishes to move left and rb wishes to move right. We temporarily place rb at the former position of b and try and place lb at its optimal position. Of course the problem is that some of the "in" constraints may be violated (since lb wishes to move left the "out" constraints cannot be violated). We remedy this with a call to *merge_left(lb)*. The placement of rb is totally symmetric, although we must first allow for the possibility that rb has been merged so we must update it's reference to the (possibly new) container of $right(c)$ and place it back at its desired position. The code for *merge_right* has not been included since it is symmetric to that of *merge_left*. We have also omitted references to the "out" constraint priority queues used by *merge_right*. These are managed in an identical fashion to "in" constraints.

Example 3. Consider the case of Example 2. The result of *satisfy_VPSC* is shown in Figure 3(d). The Lagrange multipliers calculated for c_1, c_2, c_3 are 1.333, 2.333, and -0.333 respectively. We should split on constraint c_3. We break block ABCD into ABC and D, and placing them at their optimal positions leads to positions shown in Figure 3(c). Since there is no overlap the algorithm terminates.

Theorem 7. *Let θ be the assignment to the variables V returned by* solve_VPSC(V, C). *Then θ is an optimal solution to the VPSC Problem with variables V and constraints C*

Proof. See Appendix B

Termination of *solve_VPSC* is a little more problematic. *solve_VPSC* is an example of an active-set approach to constrained optimization [14]. In practice such methods are fast and lend themselves to incremental re-computation but unfortunately, they may have theoretical exponential worst case behavior and at least in theory may not terminate if the original problem contains constraints that are redundant in the sense that the set of equality constraints corresponding to the separation constraints C, namely $\{u + a = v \mid (u + a \leq v) \in C\}$, contains redundant constraints. Unfortunately, our algorithm for constraint generation may generate equality-redundant constraints. We could remove such redundant separation constraints in a pre-processing step by adding ϵ^i to the gap for the i^{th} separation constraint or else use a variant of lexico-graphic ordering to resolve which constraint to make active in the case of equal violation. We can then show that cycling cannot occur. In practice however we have never found a case of cycling and simply terminate the algorithm after a maximum number of splits.

5 Results

We have compared our method[4] **SAT** = *satisfy_VPSC* and **SOL** = *solve_VPSC* versus **FSA**, the improved Push-Force Scan algorithm [7] and **QP** quadratic programming optimization using the Mosek solver [15]. For SAT, SOL and QP we compare with (**_OO**) and without orthogonal ordering constraints. We did not compare empirically versus the Voronoi centering algorithm [10] since it gives very poor results.

Figure 5 gives running times and relative displacement from original position for the different methods on randomly generated sets of overlapping rectangles. We varied the number of rectangles generated but adjusted the size of the rectangles to keep k (the average number of overlaps per rectangle) appoximately constant ($k \approx 10$).

We can see that FSA produces the worst displacements, and that SAT produces very good displacements almost as good as the optimal produced by SOL and QP. We can see that SAT (with or without orthogonal ordering constraints) scales better than FSA. While both SOL and QP are significantly slower, SOL is an order of magnitude faster than QP in the range tested. Adding orthogonal ordering constraints seems to simplify the problem somewhat and SOL_OO requires less splitting than SOL while QP requires more processing time to handle extra constraints. Therefore SOL_OO is significantly faster than QP_OO and SAT_OO returns a solution very near to the optimal while remaining extremely fast. Overall these results show us that SAT is the fastest of all algorithms and gives very close to optimal results.

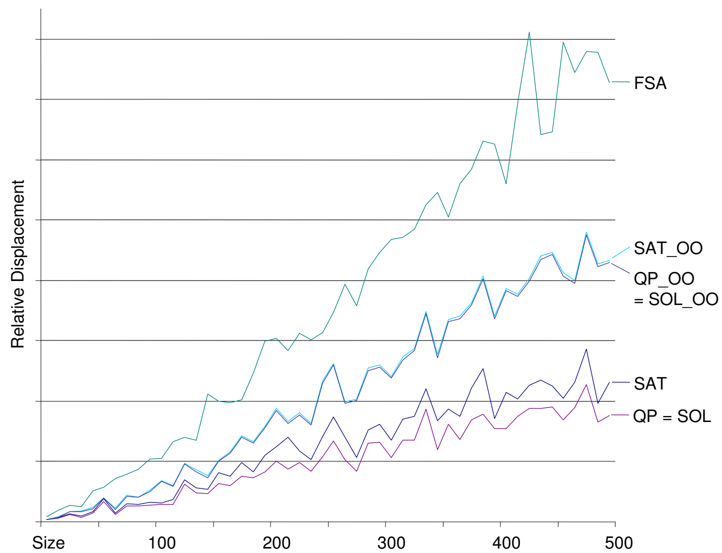
6 Example layouts

Figure 6 shows the initial layout, and the results of the various node adjustment algorithms for one of the examples. There is little difference between the SAT and SOL results. We include a SOL result with the orthogonal ordering (SOL_OO) constraints which attacks the same problem as FSA. Clearly FSA produces much more spreadout layout. Lastly the Voronoi diagram approach loses most of the structure of the original layout.

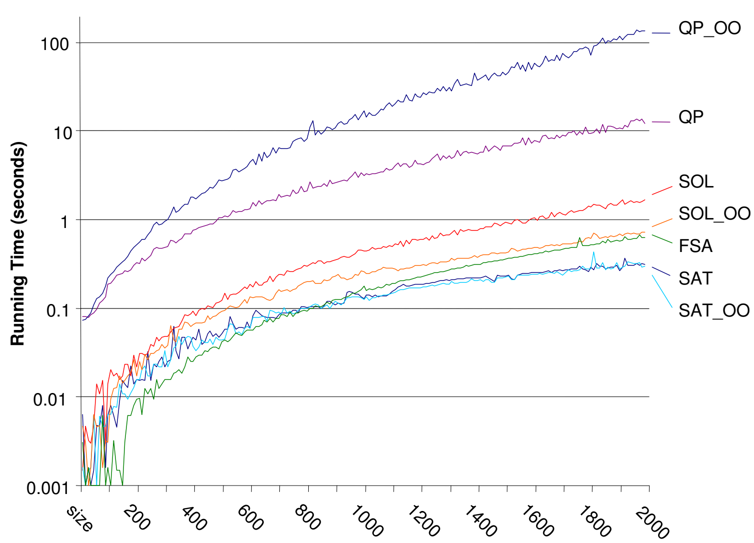
7 Conclusion

We have studied the problem of layout adjustment for graphs in which we wish to remove node overlapping while preserving the graph's original structure by moving nodes as little as possible.

[4] C++ implementation of this algorithm is available from `http://www.csse.monash.edu.au/~tdwyer`.

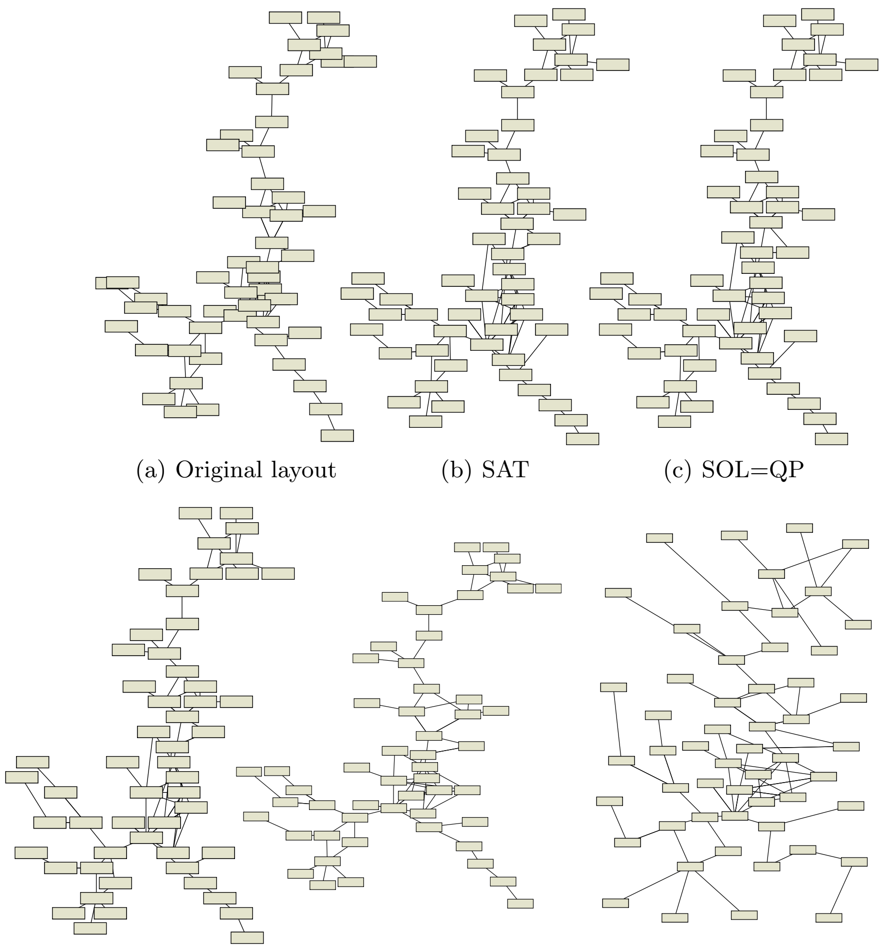


(a) Total displacement from original positions



(b) Running time

Fig. 5. Empirical comparison of various overlap removal methods



(a) Original layout (b) SAT (c) SOL=QP

(d) SOL_OO=QP_OO (e) FSA (f) Voronoi

Fig. 6. An example graph layout adjusted using various techniques.

Here, we have only considered minimizing the change in node location. Another avenue for experimentation would be to also consider minimizing the overall area of the graph, or its width or height, in order to reduce the size of the window required to display the graph.

One simple heuristic for minimizing the area is to add additional separation constraints C^{bound} that ensure all nodes are inside a bounding rectangle centered on a new point (x, y) and with height h and width w and then to solving the constrained optimization problem

$$\text{minimize } h + w + k \cdot \phi_{change} \text{ subject to } C^{no} \wedge C^{bound} \tag{1}$$

where $k \geq 0$ is a weighting factor.

Finally, we believe our results are not only interesting for layout adjustment of graphs, but may also suggest approaches to the layout of other classes of diagrams. For instance, they may suggest techniques for laying out non-overlapping windows, state charts or directed acyclic graphs.

References

1. Battista, G.D., Eades, P., Tamassia, R., Tollis, I.G.: Graph Drawing: Algorithms for the Visualization of Graphs. Prentice Hall (1999)
2. Harel, D., Koren, Y.: Drawing graphs with non-uniform vertices. In: Proceedings of the Working Conference on Advanced Visual Interfaces (AVI'02), ACM Press (2002) 157–166
3. Friedrich, C., Schreiber, F.: Flexible layering in hierarchical drawings with nodes of arbitrary size. In: Proceedings of the 27th conference on Australasian computer science (ACSC2004). Volume 26., Australian Computer Society (2004) 369–376
4. Marriott, K., Moulder, P., Hope, L., Twardy, C.: Layout of bayesian networks. In: Twenty-Eighth Australasian Computer Science Conference (ACSC2005). Volume 38 of CRPIT., Australian Computer Society (2005) 97–106
5. Misue, K., Eades, P., Lai, W., Sugiyama, K.: Layout adjustment and the mental map. Journal of Visual Languages and Computing **6** (1995) 183–210
6. Marriott, K., Stuckey, P., Tam, V., He, W.: Removing node overlapping in graph layout using constrained optimization. Constraints **8** (2003) 143–171
7. Hayashi, K., Inoue, M., Masuzawa, T., Fujiwara, H.: A layout adjustment problem for disjoint rectangles preserving orthogonal order. In: GD '98: Proceedings of the 6th International Symposium on Graph Drawing, London, UK, Springer-Verlag (1998) 183–197
8. Lai, W., Eades, P.: Removing edge-node intersections in drawings of graphs. Inf. Process. Lett. **81** (2002) 105–110
9. Gansner, E.R., North, S.C.: Improved force-directed layouts. In: GD '98: Proceedings of the 6th International Symposium on Graph Drawing, London, UK, Springer-Verlag (1998) 364–373
10. Lyons, K.A.: Cluster busting in anchored graph drawing. In: CASCON '92: Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research, IBM Press (1992) 327–337
11. Li, W., Eades, P., Nikolov, N.: Using spring algorithms to remove node overlapping. In: Proceedings of the Asia-Pacific Symposium on Information Visualisation (APVIS2005). Volume 45 of CRPIT., Australian Computer Society (2005) 131–140

12. Preparata, F.P., Shamos, M.I. In: Computational Geometry. Springer (1985) 359–365
13. Weiss, M.A.: Data Structures and Algorithm Analysis in Java. Addison Wesley Longman (1999)
14. Fletcher, R.: Practical Methods of Optimization. Chichester: John Wiley & Sons, Inc. (1987)
15. ApS, M.: (Mosek optimisation toolkit v3.2) `www.mosek.com`.

APPENDIX

A Lagrange multipliers and optimal solutions

Recall that if we are minimizing a function F with a set of convex equalities C over variables X, then we can associate a variable λ_c called the Lagrange multiplier with each $c \in C$. Given a solution $x^\star$ to C we have that this is a locally minimal solution iff there exist values for the Lagrange multipliers satisfying

$$\frac{dF}{dx}(\boldsymbol{x}^\star) = \sum_{c \in C} \lambda_c \frac{dc}{dx}(\boldsymbol{x}^\star) \tag{2}$$

for each variable $x \in X$ [14]. Furthermore, if we also allow inequalities then the above statement continues to hold as long as $\lambda_c \geq 0$ for all inequalities c of form $t \geq 0$. By definition an inequality c which is not active has $\lambda_c = 0$.

In our context we are minimizing $F = \sum_{i=1}^{n} v_i.weight \times (v_i - v_i.des)^2$ and so $\frac{\partial F}{\partial v_i} = 2 \times v_i.weight \times (v_i - v_i.des)$ for all $1 \leq i \leq n$. A constraint c has form $v - u - a \geq 0$, and so $\frac{\partial c}{\partial v} = 1$ and $\frac{\partial c}{\partial u} = -1$.

Thus Equation (2) reduces to the following requirement on each variable v_i,

$$\frac{\partial F}{\partial v_i} = \sum_{c \in in(v_i)} \lambda_c - \sum_{c \in out(v_i)} \lambda_c \tag{3}$$

Because of the simple nature of the separation constraints it is possible to compute the Lagrange multipliers efficiently and simply.

The following formula allows us to compute λ_c (more exactly $\lambda_c/2$) for the active constraints in each block in linear time. We simply perform a depth-first traversal of the constraints in *b.active* summing $v.weight \times (posn(v) - v.des)$ for the variables below this v in the tree.

Lemma 1. *If constraint c is an active constraint in some block b then*

$$\lambda_c \quad = \quad - \sum_{v \in left(b,c)} \frac{\partial F}{\partial v} \quad = \quad \sum_{v \in right(b,c)} \frac{\partial F}{\partial v}$$

If c is not active then $\lambda_c = 0$.

Proof. (Sketch) Since a block is placed at its optimal position we have that

$$\sum_{v \in b.vars} \frac{\partial F}{\partial v} = \sum_{v \in left(b,c)} \frac{\partial F}{\partial v} + \sum_{v \in right(b,c)} \frac{\partial F}{\partial v} = 0 \tag{4}$$

Hence

$$-\sum_{v \in left(b,c)} \frac{\partial F}{\partial v} = \sum_{v \in right(b,c)} \frac{\partial F}{\partial v}$$

We now show that the above formula for the Lagrange multipliers satisfies Equation 3. We must prove that for each variable $u \in b.vars$

$$\frac{\partial F}{\partial u} = \sum_{c \in in(u)} \lambda_c - \sum_{c \in out(u)} \lambda_c$$

By assumption $\sum_{c \in in(u)} \lambda_c - \sum_{c \in out(u)} \lambda_c$ equals

$$-\sum_{c \in in(u) \cap b.active} \sum_{v \in left(b,c)} \frac{\partial F}{\partial v} - \Big(\sum_{c \in out(u) \cap b.active} \sum_{v \in right(b,c)} \frac{\partial F}{\partial v}\Big)$$

But these sets of variables span $b.vars \setminus \{u\}$, and so this is equal to

$$-\sum_{v \in b.vars \setminus \{u\}} \frac{\partial F}{\partial v}$$

From Equation 4,

$$-\sum_{v \in b.vars \setminus \{u\}} \frac{\partial F}{\partial v} = \frac{\partial F}{\partial u}$$

and so these satisfy Equation 3.

B Proofs

Theorem 1. *The procedure generate_$C_x^{no}(V)$ has worst-case complexity $O(|V| \cdot k(\log|V| + k)$ where k is the maximum number of nodes overlapping a single node with appropriate choice of heap data structure. Furthermore, it will generate $O(k \cdot |V|)$ constraints.*

Proof. Sorting the nodes has $O(|V| \log |V|)$ complexity, the main for loop will be executed $2|V|$ times. First consider an *open* event. At each step *get_left_nbours* and *get_right_nbours* have $O(k \log |V|)$ complexity assuming *next_left* and *next_right* have $O(V)$ complexity and that they return a list of nodes. Note that by construction the nodes will be sorted by horizontal position. We assume that $left[v]$ and $right[v]$ are also list of nodes sorted by the nodes horizontal position. Thus *set_nbours*$(v, leftv, rightv)$ will have $O(k^2)$ complexity since the length of $leftv$, *rightv*, $left[u]$, $right[u]$ is $O(k)$. In the case of a *close* event the complexity is $O(k^2)$. Thus the total complexity of *generate_*C_x^{no} is $O(|V| \log |V|)$ plus $O(|V| \cdot (k \log |V| + k^2))$.

Theorem 5. *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 6. *The procedure satisfy_VPSC(V, C) has worst-case 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.

Theorem 7. *Let θ be the assignment to the variables V returned by* solve_VPSC(V, C). *Then θ is an optimal solution to the VPSC Problem with variables V and constraints C.*

Proof. (Sketch) For *solve_VPSC*(V, C) to terminate there must be no $c \in C$ with $\lambda_c < 0$. From Equation 2 we have that the solution at termination must be a stationary point and since the cost function has convex quadratic form this solution must be optimal.

Theorem 8. *The procedure solve_VPSC(V, C) will always terminate if C does not contain any equality-redundant constraints.*

Proof. (Sketch) Each iteration of the main loop has an associated configuration consisting of the set of active constraints in all of the blocks. There are only a finite number of such configurations. Now each iteration produces a new configuration which reduces the overall objective function as long as lb or rb are moved slightly. This will always happen unless the constraint c removed is redundant, in which case lb and rb will immediately join together in essentially the same block b but with a different active set. But from assumption this cannot happen.