Incremental Grid-like Layout Using Soft and Hard Constraints

Steve Kieffer, Tim Dwyer, Kim Marriott, and Michael Wybrow

Caulfield School of Information Technology,
Monash University, Caulfield, Victoria 3145, Australia,
National ICT Australia, Victoria Laboratory,
{Steve.Kieffer,Tim.Dwyer,Kim.Marriott,Michael.Wybrow}@monash.edu

Abstract. We explore various techniques to incorporate grid-like layout conventions into a force-directed, constraint-based graph layout framework. In doing so we are able to provide high-quality layout—with predominantly axis-aligned edges—that is more flexible than previous grid-like layout methods and which can capture layout conventions in notations such as SBGN (Systems Biology Graphical Notation). Furthermore, the layout is easily able to respect user-defined constraints and adapt to interaction in online systems and diagram editors such as Dunnart.

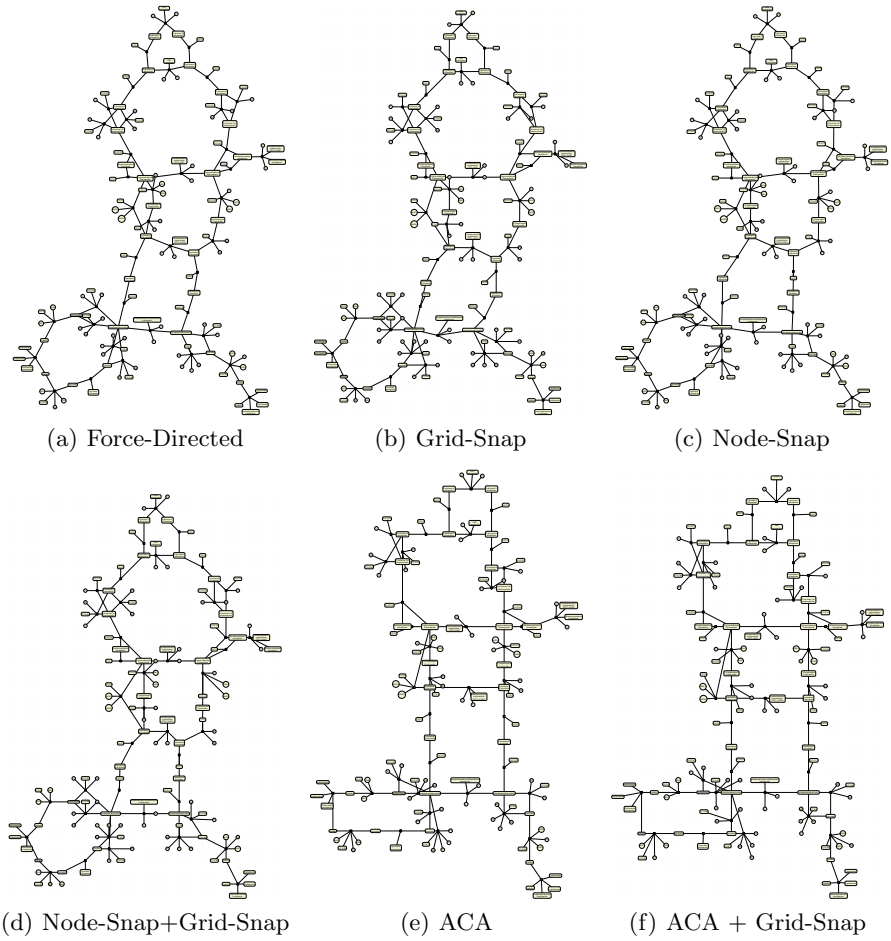
Keywords: constraint-based layout, grid layout, interaction, diagram editors

1 Introduction

Force-directed layout remains the most popular approach to automatic layout of undirected graphs. By and large these metheds untangle the graph to show underlying structure and symmetries with a layout style that is organic in appearance [2]. Constrained graph layout methods extend force-directed layout to take into account user-specified constraints on node positions such as alignment, hierarchical containment and non-overlap [5]. These methods have proven a good basis for semi-automated graph layout in tools such as Dunnart [7] that allow the user to interactively guide the layout by moving nodes or adding constraints.

However, when undirected graphs (and other kinds of diagrams) are drawn by hand it is common for a more grid-like layout style to be used. Grid-based layout is widely used by graphic designers and it is common in hand-drawn biological networks and metro-map layouts. Previous research has shown that grid-based layouts are more memorable than unaligned placements [13]. Virtually all diagram creation tools provide some kind of snap-to-grid feature.

In this paper we investigate how to modify constrained force-directed graph layout methods [5] to create more orthogonal and grid-like layouts with a particular focus on interactive applications such as Dunnart. In Figure 1 we show undirected graphs arranged with our various layout approaches compared with traditional force-directed layout.



(a) Force-Directed (b) Grid-Snap (c) Node-Snap

(d) Node-Snap+Grid-Snap (e) ACA (f) ACA + Grid-Snap

Fig. 1: Different combinations of our automatic layout techniques for grid-like layout compared with standard force-directed layout. The layout is for an SBGN (Systems Biology Graphical Notation) diagram of the Glycolysis-Gluconeogenesis pathway obtained from MetaCrop [14]. In SBGN diagrams, *process nodes* represent individual chemical reactions which typically form links in long metabolic pathways, and are often connected to several degree-1 nodes representing "currency molecules" like ATP and ADP, while precisely two of their neighbours are degree-2 nodes representing principal metabolites. It is conventional that the edges connecting main chemicals and process nodes be axis-aligned in long chains, but not the leaf edges. We achieve this by tailoring the cost functions discussed in §4.

Before proceeding, it is worth defining what we mean by a *grid-like layout*. It is commonly used to mean some combination of the following properties:

1. nodes are positioned at points on a fairly coarse grid;
2. edges are simple horizontal or vertical lines or in some cases 45° diagonals;
3. nodes of the same kind are horizontally or vertically aligned;
4. edges are orthogonal, i.e., any bends are 90°.

and thus is more general than the notion of a *grid layout*. In this paper we are primarily interested in producing layouts with properties (1) and (2), though our methods could also achieve (3). We do not consider edges with orthogonal bends, though this could be an extension or achieved through a routing post-process (a simple example of this is provided in the Appendix).

The standard approach to extending force-directed methods to handle new aesthetic criteria is to add extra "forces" which push nodes in order to satisfy particular aesthetics. One of the most commonly used functions is *stress* [9]. Our first contribution (§3) is to develop penalty terms that can be added to the stress function to reward placement on points in a grid (Property 1) and to reward horizontal or vertical node alignment and/or horizontal or vertical edges (Property 2 or 3). We call these the *Grid-Snap* and *Node-Snap* methods respectively.

However, additional terms can make the goal function rich in local minima that impede convergence to a more aesthetically pleasing global minimum. Also, such "soft" constraints cannot guarantee satisfaction and so layouts in which nodes are *nearly-but-not-quite aligned* can occur. For this reason we investigate a second approach based on constrained graph layout in which *hard* alignment constraints are automatically added to the layout so as to ensure horizontal or vertical node alignment and thus horizontal or vertical edges (Property 2 or 3). This *adaptive constrained alignment (ACA)* method (§4) is the most innovative contribution of our paper.

In §6 we provide an empirical investigation of the speed of these approaches and the quality of layout with respect to various features encoding what we feel are the aesthetic criteria important in grid-like network layout.

While the above approaches can be used in once-off network layout, our original motivation was for interactive-layout applications. In §5 we discuss an interaction model based on the above for the use of grid-like layout in interactive semi-automatic layout tools such as Dunnart.

Related Work: Our research is related to proposals for automatic grid-like layout of biological networks [1, 12, 10]. These arrange biological networks with grid coordinates for nodes in addition to various layout constraints. In particular Barsky *et al.* [1] consider alignment constraints between biologically similar nodes and Kojima *et al.* [10] perform layout subject to rectangular containers around functionally significant groups of nodes (e.g., metabolites inside the nucleus of a cell). In general they use fairly straight-forward simulated annealing or simple incremental local-search strategies. Such methods work to a degree but are slow and may never reach a particularly aesthetically appealing minimum.

Another application where grid-like layout is an important aesthetic is automatic Metro-map layout. Stott *et al.* [16] use a simple local-search ("hill-climbing") technique to obtain layout on grid points subject to a number of constraints, such as octilinear edge orientation. Wang and Chi [18] seek similar layout aesthetics but using continuous non-linear optimization subject to octilinearity and planarity constraints. This work, like ours, is based on a quasi-newton optimization method, but it is very specific to metro-map layout and it is not at all clear how these techniques could be adapted to general-purpose interactive diagramming applications.

Another family of algorithms that compute grid-like layout are so-called *orthogonal* graph drawing methods. There have been some efforts to make these incremental, for example Brandes *et al.* [3] can produce an orthogonal drawing of a graph that respects the topology for a given set of initial node positions. Being based on the "Kandinski" orthogonal layout pipeline, extending such a method with user-defined constraints such as alignment or hierarchical containment would require non-trivial engineering of each stage in the pipeline. There is also a body of theoretical work considering the computability and geometric properties of layout with grid-constraints for various classes of graphs, e.g. [4]. Though interesting in its own right, such work is usually not intended for practical application, which is the primary concern of this paper.

There are several examples of the application of soft-constraints to layout. Sugiyama and Misue [17] augment the standard force-model with "magnetic" edge-alignment forces. Ryall *et al.* [15] explored the use of various force-based constraints in the context of an interactive diagramming editor. It is the limitations of such soft constraints (discussed below) which prompt the development of the techniques described in §4.

2 Aesthetic criteria

Throughout this paper we assume that we have a graph $G = (V, E, w, h)$ consisting of a set of nodes V, a set of edges $E \subseteq V \times V$ and w_v, h_v are the width and height of node $v \in V$. We wish to find a straight-line 2D drawing for G. This is specified by a pair (x, y) where (x_v, y_v) is the centre point of each $v \in V$.

We quantify grid-like layout quality through the following metrics. In subsequent sections we use these to develop soft and hard constraints that directly or indirectly aim to optimise them. We also use these metrics in our evaluation §6.

Embedding quality We measure this using the *P-stress* function [8], a variant of *stress* [9] that does not penalise unconnected nodes being more than their desired distance apart. It measures the separation between each pair of nodes $u, v \in V$ in the drawing and their *ideal distance* d_{uv} proportional to the graph theoretic path between them:

$$\sum_{u<v\in V} w_{uv}\left((d_{uv} - d(u,v))^+\right)^2 + \sum_{(u,v)\in E} wp\left((d(u,v) - d_L)^+\right)^2$$

where $d(u, v)$ is the Euclidean distance between u and v, $(z)^+ = z$ if $z \geq 0$ otherwise 0, d_L is an ideal edge length, $wp = \frac{1}{d_L}$, and $w_{uv} = \frac{1}{d_{uv}^2}$.

Edge crossings The number of edge crossings in the drawing.
Edge/node overlap The number of edges intersecting a node box. With straight-line edges this also penalises coincident edges.[1]
Angular resolution Edges incident on the same node have a uniform angular separation. Stott *et al.* [16] give a useful formulation:

$$\sum_{v \in V} \sum_{\{e_1,e_2\} \in E} |2\pi/\mathit{degree}(v) - \theta(e_1, e_2)|$$

Edge obliqueness We prefer horizontal or vertical edges and then—with weaker preference—edges at a 45° orientation. Our precise metric is $M\left|\tan^{-1}\frac{y_u - y_v}{x_u - x_v}\right|$ where $M(\theta)$ is an "M-shaped function" over $[0, \pi/2]$ that highly penalizes edges which are almost but not quite axis-aligned and gives a lower penalty for edges midway between horizontal and vertical[2]. Other functions like those of [16, 10] could be used instead.
Grid placement Average of distances of nodes from their closest grid point.

3 Soft-Constraint Approaches

In this section we describe two new terms that can be combined with the *P-stress* function to achieve more grid-like layout: *NS-stress* for "node-snap stress" and *GS-stress* for "grid-snap stress." An additional term *EN-sep* gives good separation between nodes and edges. Layout is then achieved by minimizing

$$\text{P-stress} + k_{ns} \cdot \text{NS-stress} + k_{gs} \cdot \text{GS-stress} + k_{en} \cdot \text{EN-sep}$$

where $k_{ns,gs,en}$ control the "strength" of the various components. These extra terms, as defined below, tend to make nodes lie on top of one another. It is essential to avoid this by solving subject to node-overlap prevention constraints, as described in [6]. To obtain an initial "untangled" layout we run with $k_{ns} = k_{gs} = k_{en} = 0$ and without non-overlap constraints (Fig. 1(a)), and then run again with the extra terms and constraints to perform "grid beautification".

Minimization of the *NS-stress* term favours horizontal or vertical alignment of pairs of connected nodes (Figs. 1(c) and 5). Specifically, taking σ as the distance at which nodes should snap into alignment with one another, we define:

$$\text{NS-stress} = \sum_{(u,v) \in E} q_\sigma(x_u - x_v) + q_\sigma(y_u - y_v) \quad \text{where } q_\sigma(z) = \begin{cases} z^2/\sigma^2 & |z| \le \sigma \\ 0 & \text{otherwise} \end{cases}$$

We originally tried several other penalty functions which turned out not to have good convergence. In particular any smooth function with local maxima

[1] Node/node overlaps are also undesirable. We avoid them completely by using hard non-overlap constraints [6] in all our tests and examples.
[2] Note that $[0, \pi/2]$ is the range of $|tan^{-1}|$. The "M" function is zero at 0 and $\pi/2$, a small value $p \ge 0$ at $\pi/4$, a large value $P > 0$ at δ and $\pi/2 - \delta$ for some small $\delta > 0$, and linear in-between.

at $\pm\sigma$ must be concave-down somewhere over the interval $[-\sigma, \sigma]$, and while differentiability may seem intuitively desirable for quadratic optimization it is in fact trumped by downward concavity, which plays havoc with standard step-size calculations on which our gradient-projection algorithm is based. Thus, obvious choices like an inverted quartic $(1+(z^2-\sigma^2)^2)^{-1}$ or a sum of inverted quadratics $(1+(z+\sigma)^2)^{-1}+(1+(z-\sigma)^2)^{-1}$ proved unsuitable in place of $q_\sigma(z)$. We review the step size, gradient, and Hessian formulae for our snap-stress functions in the appendix.

We designed our *GS-stress* function likewise to make the lines of a virtual grid exert a similar attractive force on nodes once within the snap distance σ:

$$\text{GS-stress} = \sum_{u \in V} q_\sigma(x_u - a_u) + q_\sigma(y_u - b_u)$$

where (a_u, b_u) is the closest grid point to (x_u, y_u) (with ties broken by favouring the point closer to the origin), see Fig. 1(b). The grid is defined to be the set of all points $(n\tau, m\tau)$, where n and m are integers, and τ is the "grid size". With *GS-stress* active it is important to set some other parameters proportional to τ. First, we take $\sigma = \tau/2$. Next, we modify the non-overlap constraints to allow no more than one node centre to be in the vicinity of any one grid point by increasing the minimum separation distance allowed between adjacent nodes to τ. Finally, we found that setting the ideal edge length equal to τ for initial force-directed layout, before activating *GS-stress*, helped to put nodes in positions compatible with the grid.

Our third term *EN-sep* is also a quadratic function based on $q_\sigma(z)$ that separates nodes and nearby axis-aligned edges to avoid node/edge overlaps and coincident edges:

$$\text{EN-sep} = \sum_{e \in E_V \cup E_H} \sum_{u \in V} q_\sigma\left((\sigma - d(u,e))^+\right),$$

where E_V and E_H are the sets of vertically and horizontally aligned edges, respectively, and the distance $d(u, e)$ between a node u and an edge e is defined as the length of the normal from u to e if that exists, or $+\infty$ if it does not. Here again we took $\sigma = \tau/2$.

In our experiments we refer to various combinations of these terms and constraints:
Node-Snap: *NS-stress*, *EN-sep*, non-overlap constraints, $k_{gs} = 0$
Grid-Snap: *GS-stress*, *EN-sep*, ideal edge lengths equal to grid size, non-overlap, constraints with separations tailored to grid size, $k_{ns} = 0$.
Node-Snap+Grid-Snap: achieves extra alignment by adding *NS-stress* to the above **Grid-Snap** recipe (i.e. $k_{ns} \neq 0$)

4 Adaptive Constrained Alignment

Another way to customize constrained force-directed layout is by adding *hard* constraints, and in this section we describe how to make force-directed layouts more grid-like simply by adding alignment and separation constraints (Fig. 1(e)).

The algorithm, which we call *Adaptive Constrained Alignment or ACA*, is a greedy algorithm which repeatedly chooses an edge in G and aligns it horizontally or vertically (see *adapt_const_align* procedure of Figure 2). It *adapts* to user specified constraints by not adding alignments that violate these. The algorithm halts when the heuristic can no longer find any acceptable alignment to apply. Since each edge is aligned at most once, there are at most $|E|$ iterations.

We tried the algorithm with three different heuristics for choosing potential alignments, which we discuss below.

Node overlaps and edge/node overlaps can be prevented with hard non-overlap constraints and the *EN-sep* soft constraint discussed in Section 3, applied either before or after the ACA process. However, coincident edges can be accidentally created and then enforced as we apply alignments if we do not take care to maintain the orthogonal ordering of nodes. If for example two edges (u, v) and (v, w) sharing a common endpoint v are both horizontally aligned, then we must maintain either the ordering $x_u < x_v < x_w$ or the opposite ordering $x_w < x_v < x_u$.

Therefore we define the notion of a *separated alignment*, written $\mathsf{SA}(u, v, D)$ where $u, v \in V$ and $D \in \{\mathbb{N}, \mathbb{S}, \mathbb{W}, \mathbb{E}\}$ is a compass direction. Applying a separated alignment means applying two constraints to the force-directed layout—one alignment and one separation—as follows:

$$\mathsf{SA}(u, v, \mathbb{N}) \equiv x_u = x_v \text{ and } y_v + \beta(u, v) \leq y_u, \qquad \mathsf{SA}(u, v, \mathbb{S}) \equiv \mathsf{SA}(v, u, \mathbb{N}),$$
$$\mathsf{SA}(u, v, \mathbb{W}) \equiv y_u = y_v \text{ and } x_v + \alpha(u, v) \leq x_u, \qquad \mathsf{SA}(u, v, \mathbb{E}) \equiv \mathsf{SA}(v, u, \mathbb{W}),$$

where $\alpha(u, v) = (w_u + w_v)/2$ and $\beta(u, v) = (h_u + h_v)/2$. (Thus for example $\mathsf{SA}(u, v, \mathbb{N})$ can be read as, "the ray from u through v points north," where we think of v as lying north of u when its y-coordinate is smaller.)

```
proc adapt_const_align(G, C, H)
  (x, y) ← cfdl(G, C)
  SA ← H(G, C, x, y)
  while SA != NULL
    C.append(SA)
    (x, y) ← cfdl(G, C)
    SA ← H(G, C, x, y)
  return (x, y, C)
```

```
proc chooseSA(G, C, x, y, K)
  S ← NULL
  cost ← ∞
  for each (u, v) ∈ E and dir. D
    if not creates_coincidence(C, x, y, u, v, D)
      if K(u, v, D) < cost
        S ← SA(u, v, D)
        cost ← K(u, v, D)
  return S
```

Fig. 2: Adaptive constrained alignment algorithm. G is the given graph, C the set of user-defined constraints, H the alignment choice heuristic, and *cfdl* the constrained force-directed layout procedure.

Alignment Choice Heuristics We describe two kinds of alignment choice heuristics: *generic*, which can be applied to any graph, and *convention-based*, which are intended for use with layouts that must conform to special conventions, for example SBGN diagrams [11]. All of our heuristics are designed according to two principles:

1. Try to retain the overall shape of the initial force-directed layout.
2. Do not obscure the graph structure by creating undesirable overlaps.

and differ only in the choice of a *cost function* K which is plugged into the procedure `chooseSA` in Figure 2. This relies on procedure `creates_coincidence` which implements the edge coincidence test described by Theorem 1. Among separated alignments which would not lead to an edge coincidence, `chooseSA` selects one of lowest cost. Cost functions may return a special value of ∞ to mark an alignment as never to be chosen.

The `creates_coincidence` procedure works by maintaining a $|V|$-by-$|V|$ array of flags which indicate for each pair of nodes u, v whether they are aligned in either dimension and whether there is an edge between them. The cost of initializing the array is $\mathcal{O}(|V|^2 + |E| + |C|)$, but this is done only once in ACA. Each time a new alignment constraint is added the flags are updated in $\mathcal{O}(|V|)$ time, due to transitivity of the alignment relation. Checking whether a proposed separated alignment would create an edge coincidence also takes $\mathcal{O}(|V|)$ time, and works according to Theorem 1. (Proof is provided in the appendix.) Note that the validity of Theorem 1 relies on the fact that we apply separated alignments $\mathsf{SA}(u, v, D)$ only when (u, v) is an edge in the graph.

Theorem 1. *Let G be a graph with separated alignments. Let u, v be nodes in G which are not yet constrained to one another. Then the separated alignment $\mathsf{SA}(u, v, \mathbb{E})$ creates an edge coincidence in G if and only if there is a node w which is horizontally aligned with either u or v and satisfies either of the following two conditions: (i) $(u, w) \in E$ while $x_u < x_w$ or $x_v < x_w$; or (ii) $(w, v) \in E$ while $x_w < x_v$ or $x_w < x_u$. The case of vertical alignments is similar.*

We tried various cost functions, which addressed the aesthetic criteria of Section 2 in different ways. We began with a *basic cost*, which was either an estimate $K_{dS}(u, v, D)$ of the change in the stress function after applying the proposed alignment $\mathsf{SA}(u, v, D)$, or else the negation of the obliqueness of the edge, $K_{ob}(u, v, D) = -obliqueness((u, v))$, as measured by the function of Section 2. In this way we could choose to address the aesthetic criteria of *embedding quality* or *edge obliqueness*, and we found that the results were similar. Both rules favour placing the first alignments on edges which are almost axis-aligned, and this satisfies our first principle of being guided as much as possible by the shape of the initial force-directed layout. See for example Figure 1.

On top of this basic cost we considered *angular resolution* of degree-2 nodes by adding a large but finite cost that would postpone certain alignments until after others had been attempted; namely, we added a fixed cost of 1000 for any alignment that would make a degree-2 node into a "bend point," i.e., would make one of its edges horizontally aligned while the other was vertically aligned. This allows long chains of degree-2 nodes to form straight lines, and cycles of degree-2 nodes to form perfect rectangles. For SBGN diagrams we used a modification of this rule based on *non-leaf degree*, or number of neighbouring nodes which are not leaves (Figs. 1(e) and 1(f)).

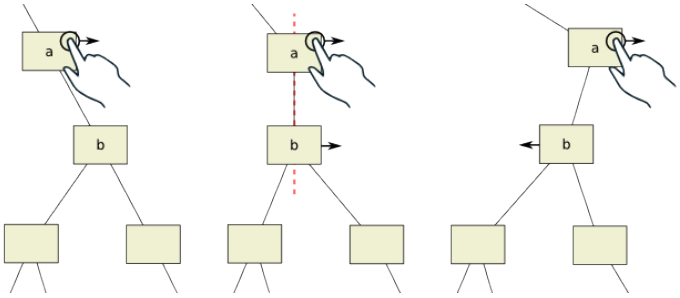


Fig. 3: Interacting with Node-Snap. The user is dragging node a steadily to the right. When the horizontal distance between a and b is less than the average width of these two nodes, the *NS-stress* function causes b to align with a. As the user continues dragging, the now aligned node b will follow until either a quick jerk of node a breaks the alignment, or else edges attached to b pull it back to the left, overcoming its attraction to a. To the user, the impression is that the alignment persisted until it was "torn" by the underlying forces in the system.

Respecting user-defined constraints Layout constraints can easily wind up in conflict with one another if not chosen carefully. In Dunnart such conflicts are detected during the projection operation described in [5], an active set method which iteratively determines the most violated constraint c and satisfies it by minimal disturbance of the node positions. When it is impossible to satisfy c without violating one of the constraints that is already in the active set, c is simply marked *unsatisfiable*, and the operation carries on without it.

For ACA it is important that user-defined constraints are never marked unsatisfiable in deference to an alignment imposed by the process; therefore we term the former *definite* constraints and the latter *tentative* constraints. We employ a modified projection operation which always chooses to mark one or more tentative constraint as unsatisfiable if they are involved in a conflict.

For conflicts involving more than one tentative constraint, we use Lagrange multipliers to choose which one to reject. These are computed as a part of the projection process. Since alignment constraints are equalities (not inequalities) the sign of their Lagrange multiplier does not matter, and a constraint whose Lagrange multiplier is maximal in absolute value is one whose rejection should permit the greatest decrease in the stress function. Therefore we choose this one.

ACA does not snap nodes to grid-points: if desired this can be achieved once ACA has added the alignment constraints by activating Grid-Snap.

5 Interaction

One benefit of the approaches described above is that they are immediately applicable for use in interactive tools where the underlying graph, the prerequisite constraint system, or ideal positions for nodes can all change dynamically. We

implemented Node-Snap, Grid-Snap and Adaptive Constrained Alignment for interactive use in the Dunnart diagram editor.[3] In Dunnart, automatic layout runs continuously in a background worker thread, allowing the layout to adapt immediately to user-specified changes to positions or constraints.

For example, Figure 3 illustrates user interaction with Node-Snap. As the user drags a node around the canvas, it may snap into alignment with an adjacent node. Slowly dragging a node aligned with other nodes will move them together and keep them in alignment, while quickly dragging a node will instead cause it to be torn from any alignments.

When we tried Node-Snap interactively in Dunnart we found that nodes tended to stick together in clumps if the σ parameter of *NS-stress* was larger than their average size in either dimension. We solved this problem by replacing the snap-stress term by

$$\sum_{(u,v)\in E} q_{\alpha(u,v)}(x_u - x_v) + q_{\beta(u,v)}(y_u - y_v)$$

where α, β are as on page 7.

In Dunnart, a dragged object is always pinned to the mouse cursor. In the case of Grid-Snap, the dragged node is unpinned and will immediately snap to a grid point on mouse-up. Other nodes, however, will snap-to or tear-away from grid points in response to changing dynamics in the layout system. During dragging we also turn off non-overlap constraints and reapply them on mouse-up. This prevents nodes being unexpectedly pushed out of place as a result of the expanded non-overlap region (§3). Additionally, since *GS-stress* holds nodes in place, we allow the user to quickly drag a node to temporarily overcome the grid forces and allow the layout to untangle with standard force-directed layout. Once it converges we automatically reapply *GS-stress*.

6 Evaluation

To evaluate the various techniques we applied each to 252 graphs from the "AT&T Graphs" corpus (`ftp://ftp.research.att.com/dist/drawdag/ug.gz`) with between 10 and 244 nodes. We excluded graphs with fewer than 10 nodes and two outlier graphs: one with 1103 nodes and one with 0 edges. We recorded running times of each stage in the automated batch process and the various aesthetic metrics described in §2, using a MacBook Pro with a 2.3GHz Intel Core I7 CPU. Details of collected data etc. are given in the Appendix.

We found that ACA was the slowest approach, often taking up to 10 times as long as the other methods, on average around 5 seconds for graphs with around 100 nodes, while the other approaches took around a second. ACA was also sensitive to the density of edges. Of the soft constraint approaches, Grid-Snap

[3] A video demonstrating interactive use of the approaches described in this paper is available at `http://www.dunnart.org`.

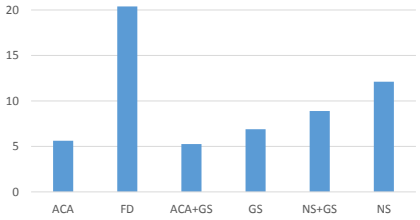


Fig. 4: Edge obliqueness (see §2) results. The hard-constraint approach ACA is better than either of the soft constraint approaches Grid-Snap (GS) and Node-Snap (NS). The combination of ACA and GS gives the best result.

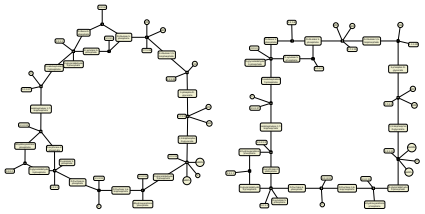


Fig. 5: Layout of a SBGN diagram of Calvin Cycle pathway shows how ACA (right) gives a more pleasing rectangular layout than Node-Snap (left).

(being very local) added very little time over the unconstrained force-directed approach.

The Edge Obliqueness (see §2) results are shown in Fig. 4 as this is arguably the metric that is most indicative of grid-like layout. Another desirable property of grid-like layout, as noted in §4 is that longer paths in the graph also be aligned. ACA does a good job of aligning such paths, as is visible in Fig. 1 and 5.

7 Conclusion

We have explored how to incorporate grid-like layout conventions into a force-directed, constraint-based graph layout framework. We give two *soft* approaches (Node-Snap, Grid-Snap) based on adding terms to the goal function, and an adaptive constraint based approach (ACA) in which *hard* alignment constraints are added greedily. We find the ACA approach is slower but gives more grid-like layout and so is the method of choice for once-off layout, at least for medium sized graphs.

We have also discussed how the approaches can be integrated into interactive diagramming tools like Dunnart. For interactive use both ACA and Grid-Snap provide good initial layouts, while Node-Snap helps the user create further alignments by hand.

Future work is to improve the speed of ACA by adding more than one alignment constraint at a time and also to use Lagrange multipliers to improve the adaptivity of ACA. One idea is to automatically reject any alignment whose Lagrange multiplier exceeds a predetermined threshold on each iteration of ACA. With this extension, running ACA continuously during interaction would allow us to achieve the behaviour illustrated in Fig. 3 through hard rather than soft constraints. Another issue with all the techniques described is the many fiddly

parameters, weights and thresholds. We intend to further investigate principled ways to automatically set these.

References

1. Barsky, A., Gardy, J.L., Hancock, R.E., Munzner, T.: Cerebral: a cytoscape plugin for layout of and interaction with biological networks using subcellular localization annotation. Bioinformatics 23(8), 1040–1042 (2007)
2. Battista, G.D., Eades, P., Tamassia, R., Tollis, I.G.: Graph drawing: algorithms for the visualization of graphs. Prentice Hall PTR (1998)
3. Brandes, U., Eiglsperger, M., Kaufmann, M., Wagner, D.: Sketch-driven orthogonal graph drawing. In: Graph Drawing. pp. 1–11. Springer (2002)
4. Chrobak, M., Payne, T.H.: A linear-time algorithm for drawing a planar graph on a grid. Information Processing Letters 54(4), 241–246 (1995)
5. Dwyer, T., Koren, Y., Marriott, K.: Ipsep-cola: An incremental procedure for separation constraint layout of graphs. Visualization and Computer Graphics, IEEE Transactions on 12(5), 821–828 (2006)
6. Dwyer, T., Marriott, K., Stuckey, P.J.: Fast node overlap removal. In: Graph Drawing. pp. 153–164. Springer (2006)
7. Dwyer, T., Marriott, K., Wybrow, M.: Dunnart: A constraint-based network diagram authoring tool. In: Graph Drawing. pp. 420–431. Springer (2009)
8. Dwyer, T., Marriott, K., Wybrow, M.: Topology preserving constrained graph layout. In: Graph Drawing, pp. 230–241. Springer Berlin Heidelberg (2009)
9. Gansner, E.R., Koren, Y., North, S.: Graph drawing by stress majorization. In: Graph Drawing. pp. 239–250. Springer (2005)
10. Kojima, K., Nagasaki, M., Jeong, E., Kato, M., Miyano, S.: An efficient grid layout algorithm for biological networks utilizing various biological attributes. BMC Bioinformatics 8(1), 76 (2007)
11. Le Novère, N., et al.: The Systems Biology Graphical Notation. Nature Biotechnology 27, 735–741 (2009)
12. Li, W., Kurata, H.: A grid layout algorithm for automatic drawing of biochemical networks. Bioinformatics 21(9), 2036–2042 (2005)
13. Marriott, K., Purchase, H., Wybrow, M., Goncu, C.: Memorability of visual features in network diagrams. Visualization and Computer Graphics, IEEE Transactions on 18(12), 2477–2485 (2012)
14. MetaCrop: `http://pgrc-35.ipk-gatersleben.de/pls/htmldb_pgrc/f?p=269:111:`
15. Ryall, K., Marks, J., Shieber, S.: An interactive constraint-based system for drawing graphs. In: Proceedings of the 10th annual ACM symposium on User interface software and technology. pp. 97–104. ACM (1997)
16. Stott, J., Rodgers, P., Martinez-Ovando, J.C., Walker, S.G.: Automatic metro map layout using multicriteria optimization. Visualization and Computer Graphics, IEEE Transactions on 17(1), 101–114 (2011)
17. Sugiyama, K., Misue, K.: Graph drawing by the magnetic spring model. Journal of Visual Languages and Computing 6(3), 217–231 (1995)
18. Wang, Y.S., Chi, M.T.: Focus+context metro maps. Visualization and Computer Graphics, IEEE Transactions on 17(12), 2528–2535 (2011)