Applying Ant Colony Optimization Metaheuristic to the DAG Layering Problem

Radoslav Andreev*, Patrick Healy*, Nikola S. Nikolov*
radoslav.andreev@ul.ie, patrick.healy@ul.ie, nikola.nikolov@ul.ie
*Department of Computer Science and Information Systems University of Limerick, IRELAND

***Abstract*— This paper[1] presents the design and implementation of an Ant Colony Optimization based algorithm for solving the DAG Layering Problem. This algorithm produces compact layerings by minimising their width and height. Importantly it takes into account the contribution of dummy vertices to the width of the resulting layering.**

I. Introduction

The Sugiyama framework [12] is the most well-known and studied heuristic for drawing directed acyclic graph (DAG). It comprises a number of steps one of which is assigning each vertex of the graph to a layer (layering step), which causes a dummy vertex to be created every time an edge crosses a layer. The next step is to order the vertices inside each layer so that the number of edge crossings between any two layers is minimised. The layering step is the one that determines what will be the height and the width of the final drawing. Usually the height represents the number of layers used to layer the graph and the width is the maximum count of real vertices in a layer. Defined like this the width ignores the contribution made by the dummy vertices. In the case where the width of a real vertex is much greater than the one of a dummy vertex, or the number of dummy vertices for any given layer is much smaller than the number of real vertices this definition is accurate enough. However, when dummy vertices are not so small compared to the real ones or when their number is high, ignoring them will inevitably result in much wider final drawing than it was expected initially. In this paper we present a layering method based on the Ant Colony Optimisation (ACO) metaheuristic [4] that minimises the width and height of the layering by taking into account the contribution of dummy vertices. To the best of our knowledge this is the first attempt to use the ACO based algorithm to layer a DAG.

In the following we introduce some preliminaries and discuss existing layering methods, followed by an introduction to the ACO metaheuristic and the representation of the layer assignment problem in its terms. Further, we describe the design and the implementation of our algorithm together with a discussion about the results achieved. Finally, a conclusion and directions for further research are given.

II. Preliminaries

A *layering* of G is a partition of V into subsets $L_1, L_2, ..., L_h$, such that if $(u, v) \in E$, where $u \in L_i$ and $v \in L_j$, then $i > j$. The *span* of an edge (u, v) with $u \in L_i$ and $v \in L_j$ is $i - j$. A layering is *proper* if all edge spans equal one. This is achieved by inserting *dummy vertices* along edges whose edge span is greater than one. The *layer span* $L(v)$ of vertex v refers to the set of layers between the topmost and the lowermost layer on which vertex v can be placed, provided that all edges point downwards. Note that the layer span for a vertex is not constant during the layering process and changes depending on the layer assignments of its neighbouring vertices.

The most common definition (without any restrictions on the layering properties) of the DAG Layering Problem (LP) can be stated as: Given a DAG, $G = (V, E)$, find a valid layer assignment so that for each vertex *u* with *y-coordinate* $y(u)$ and for each vertex *v* such that $(u, v) \in E$ the following properties are satisfied [1]:

1) $y(u)$ is an integer
2) $y(u) \geq 1$
3) $y(u) - y(v) \geq 1$

There are three important aspects of the DAG LP according to [1]:

1) The layered DAG should be compact. This means that its vertices should be evenly distributed over the drawing area.
2) The layering should be proper. This is easily achieved by inserting dummy vertices.
3) The number of dummy vertices should be small.

A layering algorithm trying to solve the DAG LP subject to additional constraints is expected to produce a layering with specified width or height or minimum number of dummy vertices or a combination of those three layering properties. The *height* of a layering is defined as the number of layers used to layer the DAG. Normally the vertices of DAGs from real-life applications have text labels and sometimes prespecified shape. Here we use a definition of *vertex width* given in [9] and stating that the width of a vertex is the width of the rectangle enclosing the vertex. If the vertex has no text label and no information about its shape or size is available we assume that its width is one unit. The *width of a layer* is defined as the sum of the widths of all vertices in that layer (including the dummy vertices) and the *width of a layering* is the maximum width of a layer [9].

The *edge density* between horizontal levels i and j with $i < j$ is defined as the number of edges (u, v) with $u \in$

[1]Supported by SFI Basic Research Grant 04/BR/CS0696

1-4244-0910-1/07/$20.00 ©2007 IEEE

$L_j \cup L_{j+1} \cup \ldots \cup L_h$ and $v \in L_1 \cup L_2 \cup \ldots \cup L_i$. The edge density of a layered DAG is the maximum edge density between adjacent layers (horizontal levels) [9]. Naturally, drawings with low edge density are more readable and easier to comprehend.

A definition of the DAG LP with additional constraints governing the width and height of the resulting layering subject to minimum number of dummy vertices is given in [11].

III. Existing layering methods

One of the most well known layering algorithms is the Longest-Path Layering (LPL) described in Algorithm 1. In the representation of this and the following algorithms we denote the set of all immediate predecessors of vertex v by $N_G^-(v)$, and the set of all immediate successors of vertex v by $N_G^+(v)$. The LPL method places all sink vertices in layer L_1, and each remaining vertex v is placed in layer L_{p+1}, where p is the longest (maximum number of edges) path from v to a sink. The attractiveness of this method is that it has linear time complexity (because the graph is acyclic) and it uses the minimum number of layers possible. The disadvantage of the LPL method is that its layerings tend to be too wide [6]. Because the area occupied by the final drawing depends on both its width and its height the Longest-Path Layering is not the best choice if minimal layering area is the main priority. Unfortunately, the problem of finding a layering with minimum width, subject to having minimum height, is NP-complete [1].

Algorithm 1 The Longest-Path Layering Algorithm

Requires: DAG $G = (V, E)$
$U \leftarrow \phi$; $Z \leftarrow \phi$
$currentLayer \leftarrow 1$
while $U \neq V$ **do**
  Select vertex $v \in V \setminus U$ with $N_G^+(v) \subseteq Z$
  if v has been selected **then**
    Assign v to the layer with a number $currentLayer$
    $U \leftarrow U \cup \{v\}$
  end if
  if no vertex has been selected **then**
    $currentLayer \leftarrow currentLayer + 1$
    $Z \leftarrow Z \cup U$
  end if
end while

Another layering method is the `MinWidth` heuristic [9] displayed in Algorithm 2. It is roughly based on the LPL algorithm. The authors employ two variables *widthCurrent* and *widthUp* to keep the width of the current layer, and the width above it, respectively. The width of the current layer, *widthCurrent*, is calculated as the number of original vertices already placed in that layer plus the number of potential dummy vertices along edges with a source in $V \setminus U$ and a target in Z (one dummy vertex per edge). The variable *widthUp* is an estimation of the width of any layer above the current one. It is the number of potential dummy vertices along edges with a source in $V \setminus U$ and a target in the current layer (one dummy vertex per edge). When a vertex is selected to be placed an additional condition `ConditionSelect` is used, which is true if v is the vertex with the maximum out-degree among the candidates to be placed in the current layer. Such a choice of v results in maximum reduction to *widthCurrent*. For a thorough discussion of the `MinWidth` heuristic the reader is referred to [9].

Algorithm 2 `MinWidth`(G)

Requires: DAG $G = (V, E)$
$U \leftarrow \phi$; $Z \leftarrow \phi$
$currentLayer \leftarrow 1$; $widthCurrent \leftarrow 0$; $widthUp \leftarrow 0$
while $U \neq V$ **do**
  Select vertex $v \in V \setminus U$ with $N_G^+(v) \subseteq Z$ and `ConditionSelect`
  if v has been selected **then**
    Assign v to the layer with a number $currentLayer$
    $U \leftarrow U \cup \{v\}$
    $widthCurrent \leftarrow widthCurrent - w_d * d^+(v) + w(v)$
    Update $widthUp$
  end if
  if no vertex has been selected OR `ConditionGoUp` **then**
    $currentLayer \leftarrow currentLayer + 1$
    $Z \leftarrow Z \cup U$
    $widthCurrent \leftarrow widthUp$
    Update $widthUp$
  end if
end while

Promote Layering (PL) [8] is a heuristic whose goal is "to develop a simple and easy to implement layering method for decreasing the number of dummy vertices in a DAG layered by some list scheduling algorithm." The PL layering method is an alternative to the network simplex method of Gansner et. al [5] but considerably easier to implement and especially useful when a commercial linear programming solver is not available. As noted PL usually runs after a layering is produced by a quick list scheduling algorithm like LPL. LPL and MinWidth on their own and in combination with the PL heuristic were the four benchmark algorithms used in this work to evaluate the performance of our ACO-based layering algorithm.

IV. Introduction to the ACO metaheuristic

Ant colonies, and more generally social insect societies, are distributed systems that, in spite of the simplicity of their individuals, represent a highly structured social organisation. As a result of this organisation, ant colonies can accomplish complex tasks that in some cases far exceed the individual capabilities of a single ant [4].

The main idea behind the Ant Colony Optimisation (ACO) metaheuristic is that self-organising principles, which allow the highly coordinated behaviour of real ants, can be exploited to coordinate populations of artificial agents that collaborate to solve computational problems.

The real ants coordinate their activities via *stigmergy*. This is a biological term about a form of indirect communication mediated by modifications of the environment. The term was first introduced by the French biologist Pierre-Paul Grasse in

1959 to refer to termite behaviour. He defined it as "Stimulation of workers by the performance they have achieved". An ant coming back to its nest from a food source it has found will deposit a chemical substance called *pheromone* which the others will find while roaming for food. By following the pheromone trail laid the rest of the ants will discover that same food source. The idea is then to use a similar artificial stigmergy, as a form of global knowledge, to coordinate societies of computational agents in an attempt to solve different combinatorial optimisation problems.

Such a computational agent is defined as "a stochastic constructive procedure that incrementally builds a solution by adding opportunely defined solution components to a partial solution under construction" [4]. Based on the above definition an ACO metaheuristic can be applied to any combinatorial optimisation problem for which a constructive heuristic can be defined. "The real issue is to find a suitable problem representation which the artificial ants will use to build their solutions" [4].

A. ACO definitions

A *tour* is a single iteration during which every ant produces a solution to the problem being solved. For a given tour all ants use the same starting point reached by the previous tour. This approach emulates a parallel work environment for all the ants comprising the colony. At the end of a tour, depending on the pheromone update strategy adopted, either one or more ants with the highest objective function value will deposit certain amounts of pheromone over the edges of the construction graph comprising its/their solution(s).

The process of constructing a solution by a single ant is called a *walk* [4]. In our algorithm each ant is placed on a randomly selected vertex v from which it starts constructing its layering. Once a vertex is assigned the next one is chosen by the ant again randomly and is assigned to a layer. This continues until each vertex is assigned to a layer.

When performing its walk an ant executes a finite number of identical actions called a *construction step* [4]. At each construction step an ant k applies a probabilistic action choice rule, called *random proportional rule* [4] given by Eq.(1), to decide which layer vertex v should be assigned to.

B. DAG LP representation in terms of the ACO metaheuristic

The first step when applying ACO to a combinatorial optimisation problem is to define the *construction graph* G_C on which ants will perform their walks. The DAG LP can be cast into the framework of the ACO metaheuristic using the construction graph $G_C = (C, H)$. Here $C = V \bigcup L$ is the set of components which includes V, the vertex set of graph G that is to be layered, and L, the set of layers. H is the set of links connecting components (vertices and layers) from C. Note that only a lower bound of $|L|$ is known beforehand but not $|L|$ itself. Each layer assignment, which consists of n couplings (v_i, l_j) of vertices and layers, corresponds to at least one ant's walk on the construction graph and cost d_{ij} is associated with every possible coupling of vertex and layer. In the original definition of the construction graph G_C, H *fully* connects the components of C [4]. This is not the case here because when assigning vertex an ant will be limited to choose from layers comprising the layer span of that particular vertex.

C. Constraints

Walks on the construction graph G_C have to satisfy the width and height constraints for the layering in order to result in a valid assignment. One particular way of generating such an assignment is by an ant's walk that randomly chooses vertex $v \in G$ as a starting point and continuing with a random selection of next vertex until all vertices of G are assigned. Additionally layers' resource capacities (width W) can be enforced by an appropriately defined neighbourhood. For example, for an ant k positioned on vertex v_i the neighbourhood N_i^k can be defined as consisting of the subset of layers of $L(v_i)$ to which v_i can be assigned without exceeding W.

D. Pheromone trails and heuristic information

Ants construct feasible solutions by iteratively adding new components to their partial solutions. While in the construction process ants repeatedly have to take the following two basic decisions:

1) choose the vertex to assign next;
2) choose the layer the vertex should be assigned to.

Pheromone trail information can be associated with any of the two decisions. It can be used to learn an appropriate order for vertex assignment or it can be associated with the desirability of assigning vertex v_i to a specific layer. In the former case, τ_{ij} represents the desirability of assigning vertex v_i immediately after vertex v_j, while in the latter it represents the desirability of assigning vertex v_i to layer l_j. In our implementation we use pheromone trail information to measure the desirability of assigning a given vertex to any of the layers from its layer span, that is, the second case. Similarly, heuristic information η_{ij} can be associated with any of the two decisions listed above but again we use heuristic information for the actual assignment and not for the order in which it is going to be done. Methods such as Breadth First Search or other similar techniques which provide a linear order of the vertices can be used to determine the order in which vertices are to be assigned. Random choice of the next vertex to be assigned is another option that may be employed.

The heuristic information can be either *static* or *dynamic*. In the static case, the values η_{ij} are computed once at the initialisation phase of the algorithm and then remain unchanged throughout the entire algorithm's run. In the dynamic case the heuristic information depends on the partial solution constructed so far and therefore has to be computed at each step of an ant's walk. Our application of the ACO to the DAG LP falls into the second category because the heuristic value $\eta_{ij} = \frac{1}{w_{ij}}$ where w_{ij} is the width of a layer $l_j \in L(v_i)$. Therefore after each assignment, which in fact moves v_i from its current layer l_{curr} to a new one l_{new}, the widths of those two layers must be changed - decreased for l_{curr} and increased for l_{new}. Moreover, the widths of the layers from

$L(v_i)$ placed between l_{curr} and l_{new} also change because of the dummy vertices induced by incoming and outgoing edges for vertex v_i. Therefore the heuristic values affected must be computed by every ant after each assignment it has made. When constructing its walk on the construction graph an ant k, that is going to assign vertex v_i, chooses layer $l_j \in L(v_i)$ with a probability given by the following equation [4]:

$$p_{ij}^k = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\Sigma_{l \in N_i^k} [\tau_{il}]^\alpha [\eta_{il}]^\beta} \quad (1)$$

Here η_{ij} is the heuristic information that is calculated a priori and τ_{ij} is the quantity of pheromone calculated as the product of the initial pheromone value, the evaporation process and the quantity deposited by ants in previous tours. The two parameters α and β determine the relative influence of the pheromone trail and the heuristic information. N_i^k is the feasible neighbourhood of ant k when assigning vertex v, that is the layer span of v. The role of α and β is the following. If $\alpha = 0$, the layers from the layer span of v with smaller widths are more likely to be selected because the influence of the pheromone information is eliminated. This corresponds to a classic stochastic greedy algorithm with multiple starting points since ants are initially randomly distributed over the vertices of the graph to be layered [4]. Conversely if $\beta = 0$, only the pheromone information is at work, and therefore the layers that had been selected by the majority of ants during previous tours that is, have accumulated high pheromone values, will more likely be selected. The absence of heuristic bias generally leads to rather poor results, and in particular, for values of $\alpha > 1$ it leads to rapid emergence of a stagnation situation where all the ants follow the same tour, which in general is strongly suboptimal [3].

E. Representing ants

There are a few key features that ants need to have in order to be able to perform their walks on the construction graph and generate feasible solutions. An ant has to be able to:

1) Memorise the partial solution it has constructed so far;
2) Determine the feasible neighbourhood for each vertex;
3) Assign a vertex to a layer subject to the height and width constraints;
4) Update the values of the heuristic matrix to reflect each new assignment;
5) Update the layer span for a vertex;
6) Compute and store the objective function value of the solution it generates;
7) Update the pheromone matrix.

The first requirement can be satisfied by storing the partial solution (walk) into an array indexed by the vertices of G and associating an integer value with each vertex representing the layer number it has been assigned to. An ant should also be able to compute the layer span of a given vertex in order to determine its neighbourhood. Additional to the layer span an ant should be able to calculate the number of dummy vertices a particular assignment would cause due to incoming edges for vertex v which cross the layers above the one to which v is assigned. These must be performed after each assignment. Finally, each ant should have a number of variables in which the characteristics of the completed layering will be kept; these are the value of the objective function, the height of the layering and the width of the layering.

V. The ACO-based layering algorithm

In our approach the graph is first layered using the fast and efficient LPL algorithm which gives the minimum number of layers graph G can be layered on. We then add a number of new layers in between the LPL ones. This results in a greater search space for the ants and gives them more freedom when constructing their solutions. Once the LPL layering is *stretched* in this manner the actual process of layering by the ant colony begins. It comprises a number of tours during which each ant builds its own layering. The layering of the best ant becomes this tour's layering. Every tour inherits the layering of its predecessor and uses it as a base to build its own. The rest of this section presents the details of our ACO-based layering algorithm.

A. Stretch LPL

The aim of this initial step of the algorithm is to add new layers to the ones introduced from LPL so that the number of layers grows to *n*, the number of vertices of G. By doing this we guarantee that no layering will be left out, that is these with the minimum width will also be in the search space. This approach also enlarges the search space, giving ants greater area for exploration. This would not be the case if they start working directly on the resulting layering from the LPL. This layering is a minimum height layering and as such it is too restrictive for ants. The only improvement they could make is to reduce the number of dummy vertices similar to the PL heuristic described in Section III. However, ants will not be able to reduce the width of the graph significantly. If we denote the number of layers produced by LPL as n_{LPL} and with *n*, the number of vertices of G, then the number of layers to add is given by $n_{nl} = n - n_{LPL}$.

One way to proceed is to add all new layers either above or below the LPL layers. Alternatively some of them can go above and the other below the LPL layers (Fig. 1).

The drawback of the both strategies is that the ant will have very few options as to where to move the vertex. Bearing in mind that each ant chooses a starting vertex randomly, it will be very limited in moving vertices without violating the initial direction of the edges of G. Of course, if the vertex is either a sink or a source an ant will have at its disposal (at least) half of all newly added layers but then it is hard to determine on which layer exactly to place the vertex as no heuristic information would be available to bias the ant's choice.

The approach suggested here is to insert the new layers in between the LPL layers. The way to do it is to divide the n_{nl} to the number of interlayer spaces from the LPL which is exactly $n_{LPL} - 1$ and then insert and re-index the layers as shown in Fig. 2.

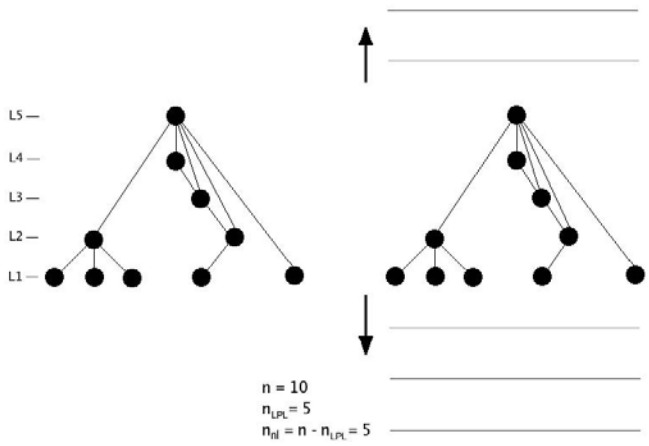

Fig. 1. The LPL and the newly added layers on top and bottom.

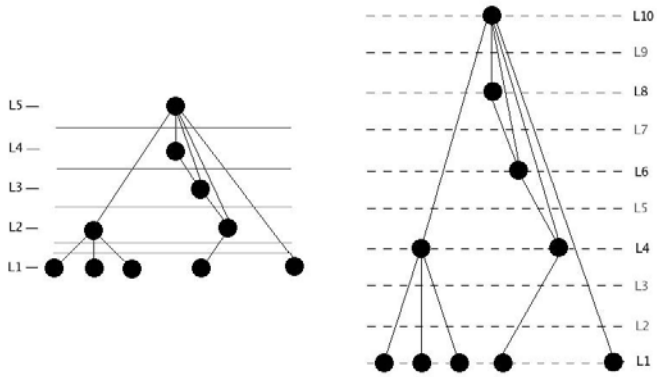

Fig. 2. The new layers inserted between the LPL ones.

By choosing this approach over the one described in Fig. 1 the layer span for each vertex is uniformly increased and therefore ants will have more possibilities for changing the layer assignment of any vertex and not only the source or sink ones. We denote the resulting (stretched) graph as G_{STR}.

Vertex width is another issue that needs careful consideration. In most real-life applications the width of dummy vertices (which in fact would be the line representing an edge) is far less than the width of a real vertex (usually a rectangle with some text inside). To reflect this the ACO-based layering algorithm allows for a variable dummy vertex width to be supplied as an initial parameter.

B. Initialisation phase

In the initial phase the input DAG $G = (V, E)$ is layered by the LPL. The resulting layering is then *stretched* as described in Section V-A allowing for a much greater exploration area for the ants. The next step is to calculate the layer span $L(v_i)$ $\forall v \in V$. Based on the layer span of a particular vertex, its corresponding elements from the heuristic matrix (one column per vertex) are initialised either to 0 or $\frac{1}{W(l_i)}$ depending on whether l_i belongs to the layer span of that vertex or not. Here $W(l_i)$ is the width of layer l_i. However all elements of the pheromone matrix are initialised to τ_0, the initial amount of pheromone laid down.

Algorithm 3 `ACO DAG LP (Initialisation phase)`

1: **Requires:** DAG $G = (V, E)$
2: $G_{LPL} \leftarrow doLPL(G)$
3: $G_{STR} \leftarrow doStretch(G_{LPL})$
{`populate the ant colony`}
4: **for** $i = 1$ to $i = \_n\_ants$ **do**
5: $\quad \_ant\_colony \leftarrow \_ant\_colony \cup ant_i$
6: **end for**
{`initialise _layer_spans;` $L(v_i)$ `is the layer span of vertex` v_i}
7: **for all** $v_i \in V$ **do**
8: $\quad \_layer\_spans[i] \leftarrow L(v_i)$
9: **end for**
{`initialise _layer_widths;` $W(l_j)$ `is the width of layer` l_j}
10: **for all** $l_j \in G_{STR}$ **do**
11: $\quad \_layer\_widths[j] \leftarrow W(l_j)$
12: **end for**
13: $\mathcal{T} \leftarrow \emptyset$; $\eta \leftarrow \emptyset$
{`column and row from` $\mathcal{T}$ `and` η `correspond to vertex and layer from` G_{STR}}
14: **for all** $\tau_{ij} \in \mathcal{T}$ **do**
15: $\quad \tau_{ij} \leftarrow \tau_0$
16: **end for**
17: **for all** $\eta_{ij} \in \eta$ **do**
18: $\quad$ **if** $l_j \in L(v_i)$ **then**
19: $\quad\quad \eta_{ij} \leftarrow \frac{1}{W(l_j)}$
20: $\quad$ **else**
21: $\quad\quad \eta_{ij} \leftarrow 0$
22: $\quad$ **end if**
23: **end for**

C. Layering phase

Once the initialisation phase is completed Algorithm 4 - Layering Phase starts. Its outermost loop runs for the specified number of tours $\_n\_tours$; 10 was the value we used in our experiments. During a single tour each ant performs its walk on the construction graph G_C and produces a layering of G_{STR}. When building its solution ant a_k repeatedly assigns vertex v_i (randomly chosen) to a layer $l_{best} \in L(v_i)$ that gives best result when executing line 6.

At line 7 the actual assignment is performed, which in turn requires that those values of the heuristic matrix η that have been affected by this particular assignment be recalculated and updated (line 8). When v_i is assigned to a layer, that is, it has been moved either up or down from its current layer, the layer

span of all neighbouring vertices of v_i changes too. Therefore the layer span for every neighbouring vertex of v_i has to be recalculated (line 10) before ant a_k picks up the next one. When a_k has assigned all vertices it is the end of its walk for the current tour. The objective function value is calculated at line 13 and stored against that ant's identifier.

At the end of a tour the evaporation step, which reduces all elements τ_{ij} of the pheromone matrix $\mathcal{T}$ by a predefined evaporation rate ρ_0, is executed. Next, the best ant for the tour a_{best} deposits pheromone on the elements τ_{ij} corresponding to its assignments. Additionally the heuristic matrix and layering of a_{best} become the starting heuristic matrix and layering for the next tour.

Algorithm 4 `ACO DAG LP (Layering phase)`

1: **Requires:** Algorithm 3 to be run first

{`begin tour`}
2: **for** $t = 1$ to $t = \_n\_tours$ **do**
3: **for all** $a_k \in \_ant\_colony$ **do**
4: {`begin ant's walk`}
5: **for all** $v_i \in V$ **do**
6: $l_{best} \leftarrow \max\left(\frac{[\tau_{ij}]^{\alpha}[\eta_{ij}]^{\beta}}{\Sigma_{l \in N_i^k}[\tau_{il}]^{\alpha}[\eta_{il}]^{\beta}}\right) \forall l_j \in L(v_i)$
7: $l_{best} \leftarrow l_{best} \cup v_i$
8: $\eta \leftarrow \eta\prime$
{`update layer span for neighbouring vertices`}
9: **for all** $u \in V$ such that $e(v_i, u) \in E$ **do**
10: $L(u) \leftarrow L\prime(u)$
11: **end for**
12: {$H(a_k)$ and $W(a_k)$ - `height and width of this ant's layering`}
13: $f(a_k) \leftarrow \left(\frac{1}{H(a_k)+W(a_k)}\right)$
14: **end for**{`end ant's walk`}
15: **end for**{`end tour`}
16: $\mathcal{T} \longleftarrow$evaporate($\mathcal{T}$)
17: $\mathcal{T} \longleftarrow a_{best}$.deposit($\mathcal{T}$)
18: $\eta \longleftarrow a_{best}(\eta)$
19: **end for**

VI. Implementation of the ACO-based layering algorithm

The algorithm was implemented in C++ with the use of the LEDA 5.0 library of efficient data types and algorithms[1]. Three classes were used, `LayeredDAG`, `Ant`, and `AntColony`. The class `LayeredDAG` inherits from LEDA's parameterised graph `GRAPH<int,int>` and has additional methods to allow for layering of the graph. The class `Ant` represents a single computational agent, which performs walks on the construction graph $G_C = (C, H)$, while building its own solution in parallel with other agents (ants). Finally, the `AntColony` class is the entity conducting the search process performed by ants.

[1]http://www.algorithmic-solutions.com/enleda.htm.

The most important method of the `Ant` class is called `performWalk()`. First it initialises this ant's pheromone and heuristic matrices, its objective function value, as well as its own copy of the layer widths data structure. Then it iterates randomly over all vertices of the graph to be layered. After a vertex is picked up, the `calcProbability()` method is called, which calculates probability values for each layer from the vertex's layer span according to the random proportional rule given by (1). The layer that corresponds to the highest probability value is chosen and the vertex is assigned to it. To accomplish this operation the algorithm invokes two other methods - `moveNode()` and `reflectNodeMovement()`.

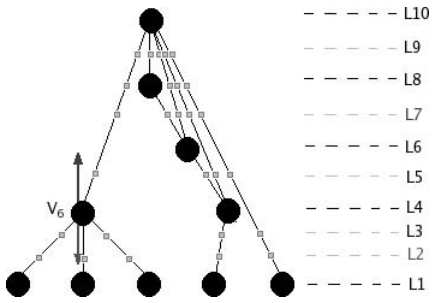

Fig. 3. Reflect vertex movement

Updating layer widths

When a vertex is moved by an ant, the widths of all layers between and including the current layer and the new one, have to be updated. The algorithm used to accomplish this is Algorithm 5. Here `_n_width` is the width of a real vertex and `_nd_width` is the width of a dummy vertex. Additionally, `outdeg`(v_i) and `indeg`(v_i) are the numbers of outgoing and incoming edges for v_i respectively. Please refer to Fig. 3 when reading the algorithm.

The main method of the `AntColony` class is `runColony()`. It calls the `performTour()` method for the specified number of tours (`i_maxIterations`), and `performTour()` calls `performWalk()` method on each ant from the colony. That method returns the objective function value the ant has achieved.

Note: When the ants produce their layering it might happen that some of the layers between the first and the last layer remain empty. To eliminate this after the layering is completed empty layers in the middle are removed and the layer numbers assigned to vertices are updated.

VII. Experiments and Results

Experiments to evaluate the performance of Ant Colony layering algorithm were conducted over a set of 1277 directed

Algorithm 5 `Updating Layer Widths`

$W(current\_layer) \leftarrow W(current\_layer) - \_n\_width$
$W(new\_layer) \leftarrow W(new\_layer) + \_n\_width$
if $new\_layer(v_i) > current\_layer(v_i)$ **then**
  for all l_j such that
  $current\_layer(v_i) \leq l_j > new\_layer(v_i)$ **do**
    $W(l_j) \leftarrow W(l_j) + outdeg(v_i) * \_nd\_width$
  end for
  for all l_j such that
  $current\_layer(v_i) < l_j \leq new\_layer(v_i)$ **do**
    $W(l_j) \leftarrow W(l_j) - indeg(v_i) * \_nd\_width$
  end for
else
  for all l_j such that
  $current\_layer(v_i) \leq l_j > new\_layer(v_i)$ **do**
    $W(l_j) \leftarrow W(l_j) + indeg(v_i) * \_nd\_width$
  end for
  for all l_j such that
  $current\_layer(v_i) > l_j \geq new\_layer(v_i)$ **do**
    $W(l_j) \leftarrow W(l_j) - outdeg(v_i) * \_nd\_width$
  end for
end if

graphs.[2]

First our algorithm was compared against the LPL algorithm and the MinWidth heuristic. Then the two were combined with the PL heuristic which in total resulted in four algorithms being used for the evaluation of our algorithm. The set of 1277 graphs was divided into 19 groups according to the number of vertices in each graph - ranging from 10 to 100 with step size 5. The main goal of these initial tests was to roughly evaluate the Ant Colony layering algorithm's performance and the feasibility of its further research. During the tests conducted four graph layering criteria namely - layering width, layering height, dummy vertices count (DVC), and maximum edge density plus a performance related one - algorithm's running time, were used.

The width of the Ant Colony layering compared with the other two algorithms is shown in Fig. 4 and Fig. 5. It can be seen that the width of the layerings produced by our algorithm is smaller than the the width of the LPL layerings and matches the ones resulting from the combination LPL plus PL heuristic. The layering width is even smaller when the dummy vertices contribution is not taken into account (the second plot in Fig. 4). This is a result of the fact that when an ant decides on which layer a vertex should be placed it uses as heuristic information the layer width estimation of all layer candidates by giving higher priority to the layers with fewer vertices currently. While it was somehow anticipated that our algorithm was going to produce narrower layerings than the LPL, the fact that it also matches the widths of the LPL plus the PL heuristic are rather encouraging. When compared with the MinWidth and MinWidth with PL our algorithm performs very close to these two algorithms especially in the case where the dummy vertices are taken into account (the first diagram in Fig. 5). Here the winner is MinWidth combined by PL followed closely by the Ant Colony layering algorithm, which in turn shows better results than the MinWidth heuristic when run on its own. This is not the case though when the contribution of dummy vertices is not taken into account (the second diagram in Fig. 5). Here clearly the winner is MinWidth followed by the MinWidth with PL and the AntColony both showing close results.

The next criteria used were the height of the layerings and the number of dummy vertices (DVC). The results are shown in Fig. 6 and Fig. 7. The clear winner when it comes to the height of the layering is of course the LPL algorithm. The Ant Colony layerings are between 20 and 30% higher than the LPL ones and this is a result of achieving smaller layering width than the LPL. One thing to note here is the fact that even by "stretching" the LPL layerings by those 20 to 30% our algorithm manages to keep the same number of dummy vertices as the original LPL layering (second diagram in Fig. 6). The Dummy Vertices Count (DVC) of the Ant Colony though is greater than the LPL when combined with PL.

The last two criteria used are the Edge Density (ED) and the Running Time (RT). ED is the maximum number of edges between any two layers of the resulting layering. The lower this value is the more uniform distribution of edges is observed in the final drawing of the graph we are layering. According to Fig. 8 and Fig. 9 the ED of the layerings resulting from applying the Ant Colony are between the values of the MinWidth and MinWidth with PL and are better when compared with the LPL and LPL with PL. When comparing the running times - as expected LPL and MinWidth are the winners. While this was no suprise to us it was encouraging to see that the RT of our algorithm is not much higher when LPL and MinWidth are combined with the PL heuristic.

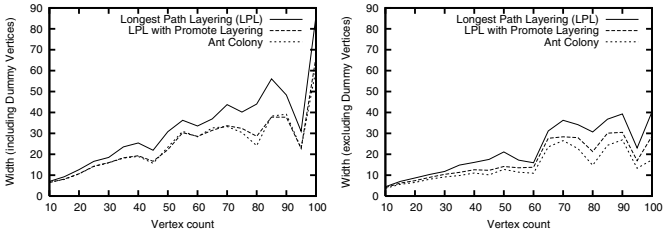

Fig. 4. Width of Ant Colony Layering Compared with LPL and LPL with PL

VIII. Parameter tuning

The Ant Colony operates depending on a number of parameters that set the number of ants, tours to be performed, initial pheromone values, rate of pheromone evaporation and so on. There are two main parameters though named α and β that influence the significance of the pheromone and heuristic information respectively when a decision is made by the ant. Various tests were performed for α and β ranging from 1 to 5 and the best results were achieved for $\alpha = 3$ and

[2]AT&T graphs available from `http://www.graphdrawing.org`

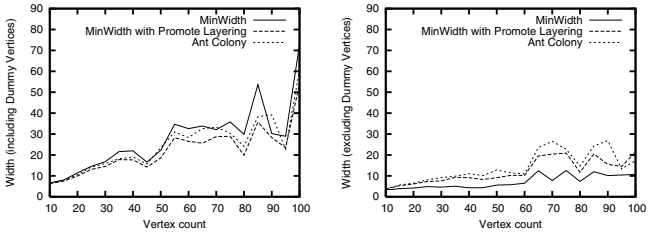

Fig. 5. Width of Ant Colony Layering Compared with MinWidth and MinWidth with PL

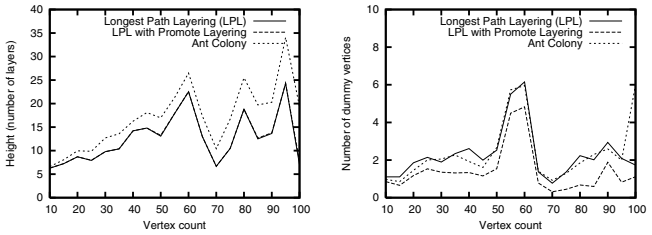

Fig. 6. Height and DVC of Ant Colony Layering Compared with LPL and LPL with PL

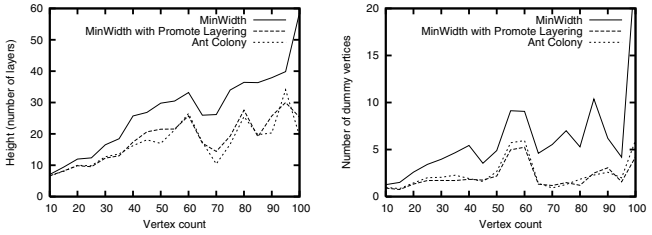

Fig. 7. Height and DVC of Ant Colony Layering Compared with MinWidth and MinWidth with PL

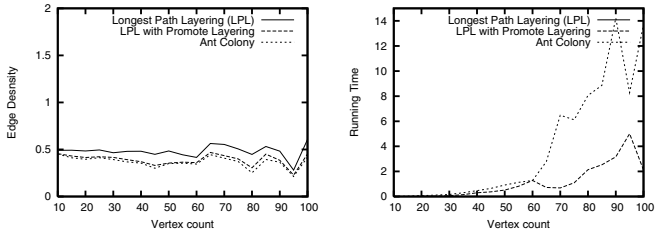

Fig. 8. Edge density and Running time of Ant Colony Layering Compared with LPL and LPL with PL

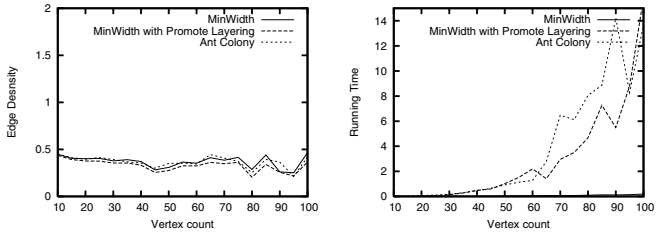

Fig. 9. Edge density and Running time of Ant Colony Layering Compared with MinWidth and MinWidth with PL

$\beta = 5$ followed closely by the results for $\alpha = 1$ and $\beta = 3$ showing slightly lower performance but at the expense of longer running times for the former. Therefore it was decided that 1 and 3 will be used as respective values for those two parameters in our further investigations. Interestingly those values for α and β showed good performance of the Ant Colony Optimization heuristics when tested with instances of the Traveling Salesman Problem (TSP) as reported previously [4] Another parameter considered is the dummy vertex width (`_nd_width`) although this is not a parameter of the Ant Colony it has, as it proved from the tests we run, a direct influence on the quality of the final layering. We run the algorithm for values for `_nd_width` ranging from 0.1 to 1.2 with step 0.1 and the best results were achieved for `_nd_width` $= 1.1$ closely followed by `_nd_width` $= 1$. Again the slightly better performance for 1.1 could not justify the longer running time and therefore the `_nd_width` $= 1$ will be used in our experiments.

IX. Conclusion

On the basis of the initial tests we can conclude that Ant Colony layering algorithm performs well when compared against the two base layering methods LPL and MinWidth alone and combined with the PL heuristic. Those two layering methods target two competing layering characteristics the height (LPL) and the width (MinWidth) of a layering. The fact that the Ant Colony layering algorithm produces comparable results (only slightly worse) in the key area (height and width respectively) for each of the two algorithms and in the same time outperforms them in the other layering criteria proves that the algorithm is doing what it is supposed to do and appears to be more universal than the other two. However, the running time of the AntColony is greater than any of the two base methods. This is due to the fact that the Ant Colony layering algorithm uses LPL to build its initial layering. Nevertheless when the Ant Colony layering algorithm is compared to LPL and MinWidth with PL the running time of the first is not significantly worse than the running time of the other two algorithms.

References

[1] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis, *Graph Drawing: Algorithms for the visualization of graphs*. New Jersey, USA, Prentice Hall, Inc., 1999.

[2] E. G. Coffman and R. L. Graham, *Optimal scheduling for two processor systems*. Acta Informatica, Volume 1, pages 200-213, 1972.

[3] M. Dorigo, V. Maniezzo, and A. Colorni, *The Ant System: Optimization by a colony of cooperating agents*. IEEE Transactions on Systems, Man, and Cybernetics Part B: Cybernetics, Volume 26, pages 29-41, 1996.

[4] M. Dorigo, T. Stützle, *Ant Colony Optimization*. Cambridge, Massachusetts and London, England, The MIT Press 2004.

[5] E. R. Gansner and E. Koutsofios and S. C. North and Kiem-Phong Vo, *A Technique for Drawing Directed Graphs*. Software Engineering, Volume 19, pages 214-230, 1993.

[6] P. Healy and N. S. Nikolov, *How to Layer a Directed Acyclic Graph*. GD '01: Revised Papers from the 9th International Symposium on Graph Drawing, pages 16-30, Springer-Verlag, 2002.

[7] G. L. Nemhauser and L. A. Wolsey, *Integer and Combinatorial Optimization*. Wiley-Interscience series in discrete mathematics and optimization, John Wiley & Sons, Inc., 1988.

[8] N. S. Nikolov and A. Tarassov *Graph layering by promotion of nodes*. journal of Discrete Applied Mathematics, issue 5 IV ALIO/EURO Workshop on Applied Combinatorial Optimization, Volume 154, pages 848-860, Elsevier, 2006.

[9] N. S. Nikolov, A. Tarassov, and J. Branke *In Search for Efficient Heuristics for Minimum-Width Graph Layering with Consideration of Dummy Nodes*. The ACM Journal on Experimental Algorithmics, Volume 10, Section: 2 - WEA'04, pages 1-27, 2005.

[10] K. Sugiyama, *Graph Drawing and Applications for software and knowledge engineers*. Series on Software Engineering and Knowledge Engineering, Volume 11, World Scientific, 2002.

[11] N. S. Nikolov, *A Polyhedral Approach to Directed Acyclic Graph Layering*. A PhD thesis, Department of Computer Science and Information Systems, University of Limerick, July 2002.

[12] K. Sugiyama, S. Tagawa, and M. Toda *Methods for Visual Understanding of Hierarchical Systems*. IEEE Trans. on Systems, Man, and Cybernetics, vol. SMC-11, no. 2, pp. 109-125, 1981.