Applying ACO Metaheuristic to the Layer Assignment Problem

Technical Report, UL-CSIS-06-1

Radoslav Andreev, Patrick Healy, Nikola S. Nikolov

Abstract

This report presents the design and implementation of Ant Colony Optimisation (ACO) based heuristic for solving the Layer Assignment Problem (LAP) for a directed acyclic graph (DAG). This heuristic produces compact layerings by trying to minimise their width and height. It takes into account the contribution of dummy vertices to the width of the resulting layering.

1 Introduction

The Sugiyama framework [12] is the most well-known and studied heuristic for drawing directed acyclic graphs. It is comprised of a number of steps one of which consists of assigning each vertex of the graph to a layer (layering step), which causes 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 (dummy vertices) 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 width is the maximal sum of real vertices in one layer. Defined like this the width ignores the contribution made by the dummy vertices. In case where the width of 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 their number is high, ignoring them will inevitably result in a final drawing that is much wider than expected initially.

In this report we present a layering method based on the Ant Colony Optimisation (ACO) metaheuristic [6] which tries to minimise 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 directed acyclic graph. In the following we introduce some preliminaries and discuss existing layering methods, followed by 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 directionits for further research are given.

2 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 layering is *proper* if all edge spans equal one. This is achieved by inserting *dummy vertices* along edges whose span is greater than one. The *span* of an edge (u, v) with $u \in L_i$ and $v \in L_j$ is $i - j$.

A layering algorithm is expected to produce a layering with specified either width and height, or aspect ratio. The *height* of a layering is the number of layers. Normally the vertices of DAGs from real-life applications have text labels and sometimes prespecified shape. We define the *width* of a vertex to be the width of the rectangle that encloses the vertex [13]. 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 usually defined as the sum of the widths of all vertices in that layer (including the dummy nodes) and the *width of a layering* is the maximum width of a layer [13].

The *edge density* between horizontal levels i and j with $i < j$ is defined as the number of edges (u, v) with $u \in L_j \cup L_{j+1} \cup ... \cup L_h$ and $v \in L_0 \cup L_1 \cup ... \cup L_i$. The edge density of a layered DAG is the maximum edge density between adjacent layers (horizontal levels) [13]. Naturally, drawings with low edge density are more readable and easier to comprehend.

The layering of an acyclic digraph $G = (V, E)$ is considered to be the second step of the Sugiyama framework for drawing general digraphs with the first being the transformation of a general digraph into acyclic. To do this one would reverse the direction of the necessary number of edges so that existing cycles are broken.

There are three important aspects of the layering problem [2]:

1. The layred digraph should be compact. It 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.

The Layer Assignment Problem (LAP) can be stated as: Given a directed acyclic graph (DAG), $G = (V, E)$, find a valid layer assignment so that for each vertex u with *y-coordinate* $y(u)$ the following properties are satisfied: [2]

1. $y(u)$ is an integer

2. $y(u) \geq 1$

3. $y(u) - y(v) \geq 1$

Another definition of the LAP can be derived from the Generalised Assignment Problem (GAP). In GAP a set of tasks $i \in I$, have to be assigned to a set of agents $j \in J$. Each agent has only a limited capacity a_j and each task i assigned to agent j consumes a quantity r_{ij} of the agent's capacity. Also, the cost d_{ij} of assigning task i to agent j is given. The objective then is to find a feasible task assignment with the minimum cost [6].

Based on the above definition, the LAP can be stated as follows:
A set of vertices $v \in V$ of a directed acyclic graph, $G = (V, E)$ has to be assigned to a set of layers $l \in L$. Every layer has only a limited 'capacity' (width) W. Every vertex v_i assigned to layer l_j consumes r_{ij} (vertex's width) amount of the layer's capacity. Also, the cost d_{ij} of assigning vertex v_i to a layer l_j is given as the number of dummy vertices, which edges incident to v_i will cause if that particular assignment is to be accomplished. The objective function then is to find a feasible layer assignment for the vertices of G with minimum cost (number of dummy vertices), while the requirement that no layer may exceed the specified width W is satisfied.

It is NP-complete to find a layering with the minimum width when the contribution of the dummy nodes is taken into account [3]. To tackle it our algorithm will comprise two steps; first a layering not exceeding some width W, given at the start, will be produced by the ant colony; second the number of dummy vertices will be minimised subject to the width from step one. Currently only the first step is implemented in the algorithm presented in this report.

Let $y_{ij} = 1$ if vertex v_i is assigned to layer l_j and $y_{ij} = 0$ otherwise. Then the LAP can be formally defined as:

$$\min \sum_{j=1}^{m} \sum_{i=1}^{n} d_{ij} y_{ij} \tag{1}$$

subject to:

$$\sum_{i=1}^{n} r_{ij} y_{ij} \leq W, \quad j = 1, \ldots, m \tag{2}$$

$$\sum_{j=1}^{m} y_{ij} = 1, \quad i = 1, \ldots, n \tag{3}$$

$$y_{ij} \in \{0, 1\}, \quad i = 1, \ldots, n, \quad j = 1, \ldots, m \tag{4}$$

The constraints in 2 implement the limited resource capacity (width) of the layers, while constraints given by 3) and 4 mandate that every vertex is assigned to exactly one layer.

The following definition is needed when defining appropriate LAP representation to be used by the Ant Colony Optimisation heuristic.
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.

3 Existing layering methods

One of the most well known layering algorithms is the Longest-Path Layering (LPL) described in Algorithm 1. It 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 [8]. Because the compactness of the final drawing depends on both the width and the height the Longest-Path Layering is not the best choice if compactness of the layering is a main priority. Unfortunately, the problem of finding a layering with minimum width, subject to having minimum height, is NP-complete [2].

Another layering method is the `MinWidth` heuristic displayed in Algorithm 2. It is roughly based on the LPL [11]. 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 $V \backslash U$ 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 $V \backslash U$ and a target

Algorithm 1 The Longest-Path Algorithm(G)

Requires: DAG $G = (V, E)$

$U \leftarrow \phi$
$Z \leftarrow \phi$
$currentLayer \leftarrow 1$
while $U \neq V$ **do**
  Select node $v \in V \setminus U$ 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 node has been selected **then**
    $currentLayer \leftarrow currentLayer + 1$
    $Z \leftarrow Z \cup U$
  end if
end while

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 thorough discussion of the `MinWidth` heuristic the reader is referred to [13].

Promote Layering (PL), Algorithm 3, is a heuristic introduced in [10] and its goal is "to develop a simple and easy to implement layering method for decreasing the number of dummy nodes in a DAG layered by some list scheduling algorithm". The PL layering method is an alternative to the network simplex method of Gansner et. all [7] 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 the LPL. LPL and MinWidth alone and in combination with the PL heuristic were the two benchmark algorithms used in this work to evaluate the performance of the ACO-based layering algorithm. Next is an algorithm that can prodice a layering with predefined width and guarantees that its height will be less than or equal to a certain threshold value, is the algorithm of Coffman-Graham [4]. It takes as an input reduced directed graph G and a positive integer W representing the desired width for the resulting layering. The product is layering of G with width at

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 node $v \in V \setminus U$ 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 node has been selected OR `ConditionGoUp` **then**
    $currentLayer \leftarrow currentLayer + 1$
    $Z \leftarrow Z \cup U$
    $widthCurrent \leftarrow widthUp$
    Update $widthUp$
  end if
end while

Algorithm 3 `PromoteNode`(v)

Require: A layered DAG $G = (V, E)$ with the layering information stored in a global node array of integers called *layering*; a node $v \in V$.

$dummydiff \leftarrow 0$
for all $u \in N_G^-(v)$ **do**
  if $layering[u] = layering[v] + 1$ **then**
    $dummydiff \leftarrow dummydiff +$ `PromoteNode`(u)
  end if
end for
$layering[v] \leftarrow layering[v] + 1$
$dummydiff \leftarrow dummydiff - N_G^-(v) + N_G^+(v)$
return $dummydiff$

most W and height $h \leq (2-2/W) * h_{min}$, where h_{min} is the minimum height of a layering of width W. We should note however that this method does not take into account the width of the dummy vertices. This is a convenient assumption but is accurate enough only when the width of the dummy vertices is considerably smaller than the size of the real vertices. When this is not the case the Coffman-Graham algorithm has to be adjusted to reflect the width of the dummy vertices [2]. This method has two phases: in the first phase it orders the vertices, and in the second it assigns them to layers.

The last existing algorithm to discuss here is the Integer Linear Programming (ILP) Algorithm of Gansner, Koutsofios, North and Vo [7]. This is an exact algorithm that can compute the minimum number of dummy vertices in polynomial time. The problem is represented by the following integer linear program:

$$\min \sum_{(u,v)\in E} l(u,\alpha) - l(v,\alpha) \tag{5}$$

subject to:

$$l(u,\alpha) - l(v,\alpha) \geq 1, \forall (u,v) \in E \tag{6}$$

$$l(u,\alpha) \geq 0, \forall u \in V \tag{7}$$

$$\text{all } l(u,\alpha) \text{ are integer} \tag{8}$$

The linear programming relaxation of this integer program has always an integer solution because its constraint matrix is totally unimodular [9]. The algorithm of Gansner et al. has not been proven polynomial but reportedly requires a few iterations and runs fast. In practise, the dummy vertex minimisation methods, such as this one, not only give shorter edge lengths and fewer dummy vertices, but also tend to give relatively compact layerings [2].

4 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 [6].

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" [1]. 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 would 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"[6]. Based on the above definition 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" [6].

4.1 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 emulate a parallel work 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 amount 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 *walk* [6]. In our algorithm ants are put randomly on a vertex v and each starts constructing its layering from it. Once vertex is assigned the next one is chosen again randomly and so forth until all vertices have layer assigned to them.

When performing its walk an ant executes a finite number of identical actions called *construction step* [6]. At each construction step an ant a_k applies a probabilistic action choice rule, called *random proportional rule* [6] given by 9, to decide, which layer vertex v should be assigned to.

4.2 LAP representation in terms of the ACO metaheuristic

The first step when applying ACO to combinatorial optimisation problem is to define the *construction graph* G_C on which ants will perform their walks. The LAP 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 construction graph G_C H *fully* connects the components of C [6]. 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.

4.3 Constraints

Walks on the construction graph G_C have to satisfy constraints 3 and 4 in order to result in a valid assignment. One particular way of generating such an assignments 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 in 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.

4.4 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 [6]:

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 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 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 LAP 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 [6]:

$$p_{ij}^k = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\Sigma_{l \in N_i^k} [\tau_{il}]^\alpha [\eta_{il}]^\beta} \tag{9}$$

Here η_{ij} is the heuristic information that is calculated a priori and τ_{ij} is the pheromone 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 [6]. 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 [5].

4.5 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 constraints (2) to (4);
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.

5 The ACO-based layering algorithm

In this approach the graph is first layered using the fast and efficient LPL algorithm. It gives the minimum number of layers graph G can be layered on and it is a good starting point for the Ant Colony layering algorithm.

5.1 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 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 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 similarly to the PL heuristic described in section 3. 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} the number of layers to add is given by $n_{nl} = n - n_{LPL}$.

One way to go 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 this approach is that ants cannot move around vertices without violating the initial direction of the edges of G. Bearing in mind that each ant will choose a vertex randomly, unless the vertex is either a sink or a source, the ant will have very limited options as to where to move the vertex. 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 on Fig. 2.

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.

Vertex width is another issue that needs a 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 (a

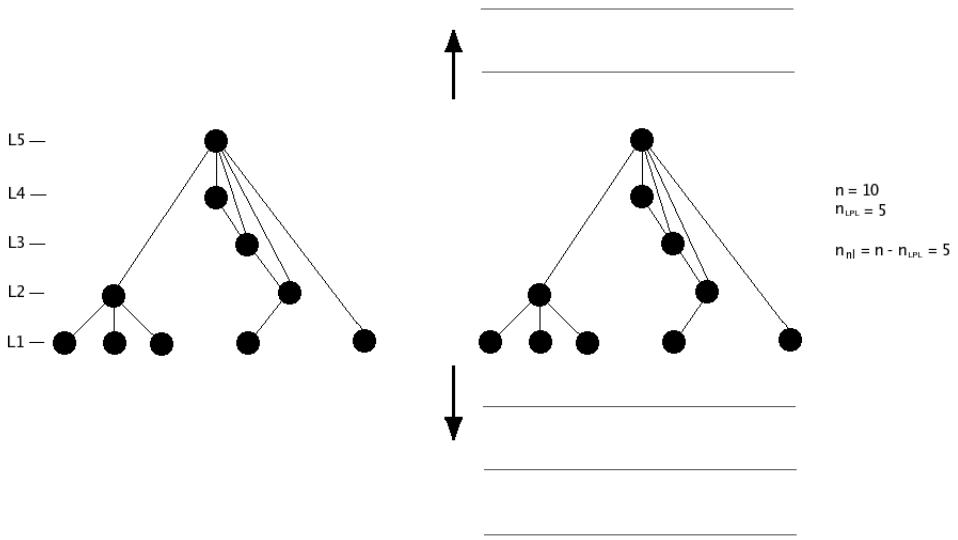

Figure 1: The LPL and the newly added layers on top and bottom.

rectangle with some text inside). To reflect this the ACO-based layering algorithm allows for a variable dummy vertex width to be supplied as initial parameter.

5.2 Initialisation phase

In this initial phase the input DAG $G = (V, E)$ is layered by the LPL. The resulting layering is then *stretched* as described in Section 5.1 allowing for a much greater exploration area for the ants. 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 4 `ACO LAP (Initialisation phase)`

Requires: DAG $G = (V, E)$

$G_{LPL} \leftarrow doLPL(G)$
$G_{STR} \leftarrow doStretch(G_{LPL})$

{`populate the ant colony`}
for $i = 1$ to $i =$ $\_n\_ants$ **do**
  $\_ant\_colony \leftarrow \_ant\_colony \cup ant_i$
end for

{`initialise _layer_spans;` $L(v_i)$ `is the layer span of vertex` v_i}
for all $v_i \in V$ **do**
  $\_layer\_spans[i] \leftarrow L(v_i)$
end for

{`initialise _layer_widths;` $W(l_j)$ `is the width of layer` l_j}
for all $l_j \in G_{STR}$ **do**
  $\_layer\_widths[j] \leftarrow W(l_j)$
end for

$\mathcal{T} \leftarrow \emptyset$; $\eta \leftarrow \emptyset$

{`column and row from` $\mathcal{T}$ `and` η `correspond to vertex and layer from` G_{STR}}
for all $\tau_{ij} \in \mathcal{T}$ **do**
  $\tau_{ij} \leftarrow \tau_0$
end for

for all $\eta_{ij} \in \eta$ **do**
  if $l_j \in L(v_i)$ **then**
    $\eta_{ij} \leftarrow \frac{1}{W(l_j)}$
  else
    $\eta_{ij} \leftarrow 0$
  end if
end for

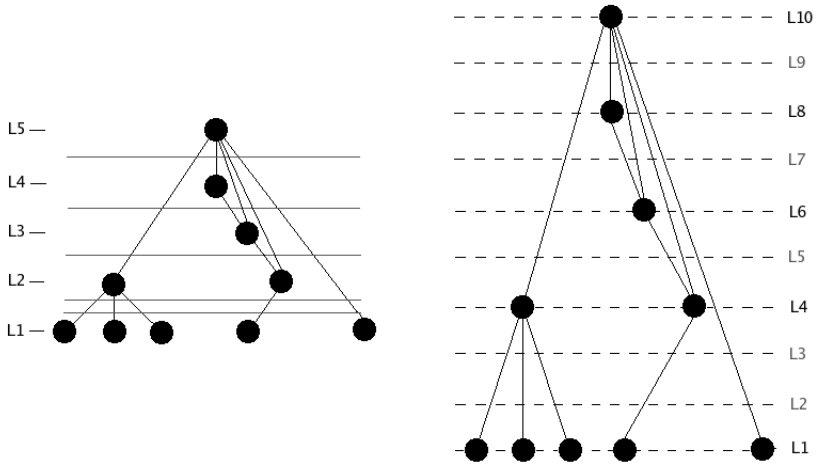

Figure 2: The new layers inserted between the LPL ones.

5.3 Layering phase

Once the initialisation phase is completed Algorithm 5 - Layering Phase starts. His outermost loop runs for the specified number of tours $\_n\_tours$. 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 turns 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, 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 next tour.

Algorithm 5 `ACO LAP (Layering phase)`

Requires: Algorithm 4 to be run first

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

6 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, `LayredDAG`,

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

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.

6.1 More on the class members

LayeredDAG:

Data members

- array<double> _layer_widths - keeps the width of each layer;
- list<node> _dummy_set - this list contains all dummy vertices;
- array<list<node> > _layer - each element from this array is a list that contains the vertices assigned to a particular layer;
- list<edge> _long_edges - the edges that span more than one layer are kept here;
- list<three_tuple<edge, node, node> > _new_edges - list of newly added edges in the format - edge, source, target. These new edges have at least one dummy vertex as either their source or target;
- int _n, _m - vertices and edges of the initial graph to be layered;
- int _n_width - width of a real vertex;
- double _dn_width - width of a dummy vertex;

Methods

- assignLayers(int layering_method) - layer the graph using the layering method specified as parameter;
- fillLayers() - fills the _layer data structure;
- fillLayerWidths() - calculate the width for each layer and stores it in _layerWidths;
- stretchGraph() - adds new layers between the existing ones as described earlier;

- `insertDummyNodes()` - inserts dummy vertices;
- `calcLayerSpan(vertex i_vertex)` - returns an array of layers comprising the layer span $L(i\_vertex)$;
- `calcLayerWidth(int i_layerNumber)` - returns a decimal number representing the width of a layer;
- `prepareDAG()` - one of the most important methods in the `LayeredDAG`. It implements the initial LPL layering, inserts the new layers as described in section 5.1 and reassigns the layer number associated with every vertex.

Ant:

Data members

- `double _objectiveFuncValue` - this ant's objective function value;
- `array<list<node> > _antLayering` - this ant's layering;
- `array<double> _antLayerWidths` - each ant uses its own copy of the `_layerWidths` data structure;
- `node_array<array<double> > _pheromoneMatrix` - this ant's pheromone matrix;
- `node_array<array<double> > _heuristicMatrix` - this ant's heuristic matrix;

It is necessary that each ant has its own heuristic matrix (`_heuristicMatrix`), layering ($\_antLayering$), and layer widths ($\_antLayerWidths$) data structures in order not to change the ones belonging to current tour while performing its walk. This way every ant is building its own solution from the same starting point, independently from the rest of the colony. However, the rationale behind keeping a local copy of the pheromone matrix (`_pheromoneMatrix`) is different, its sole purpose is to keep running time low by avoiding reading heuristic values each time from the tour's heuristic matrix - property of `AntColony` class.

Methods

- `performWalk()` - governs solution building process for this ant;

- `calculateProbability(node v)` - implements the random proportional rule given by 9. This method returns the layer number to which v is to be assigned.
- `moveNode(node v, int` $i\_oldLayer$`, int` $i\_newLayer$`)` - moves v from its current layer $i_oldLayer$ to $i_newLayer$ returned by `calculateProbability(node v)`.
- `reflectNodeMovement(node v, int` $i\_oldLayer$`, int` $i\_newLayer$`)` - updates the layer spans of all neighbouring vertices of v plus $\_antLayering$, $\_antLayerWidths$, and $\_heuristicMatrix$ data structures.
- `calcThisAntObjectiveFuncValue()` - assesses the layering produced by this ant.

AntColony:

Data members

- `int _n_ants` - number of ants comprising the colony;
- `double _initialPheromoneValue` - the elements of the pheromone matrix are initialised to this value;
- `double _alpha` - represents the relative influence of the pheromone information when ant makes its decision;
- `double _beta` - represents the relative influence of the heuristic information when ant makes its decision;
- `double _rho` - the rate of pheromone evaporation;
- `double _addPheromoneValue` - the amount of pheromone to be deposited after each tour;

Methods

- `initColony(int i_ants, double i_initialPheromoneValue, i_alpha, i_beta, ...)` - populates the colony and sets its parameters;
- `fillHeuristicMatrix(LayeredDAG &i_inputDAG, LayeredDAG &i_stretchedDAG)` - calculates the initial heuristic values for the elements of η ;
- `fillPheromoneMatrix(LayeredDAG &i_inputDAG, LayeredDAG &i_stretchedDAG)` - sets all elements of τ to `_initialPheromoneValue`;

- `void runColony(int i_MaxIterations)` - runs the colony for the specified number of tours - `i_MaxIterations`;
- `performTour()` - runs a single iteration (tour) of the algorithm;

6.2 Implementation highlights

The Ant class:

The most important method in this class is called `performWalk()`. First it initialises this ants 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 calculate probability values for each layer from the vertex's layer span according to 9. The layer corresponding 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()`. The former only removes the vertex from its current layer and adds it to the new one. The latter method does more. Its task is to update the layer span of all neighbouring vertices (of the vertex being moved); update layer widths for the layers affected (those are all layers between the current and the new layers) and finally update the column of the heuristic matrix corresponding to this vertex.

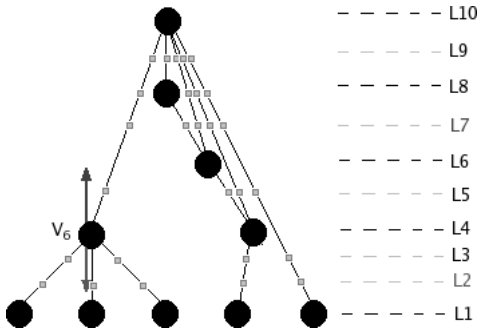

Figure 3: Reflect vertex movement

Updating layer widths

When vertex is moved by 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 6. 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.

Algorithm 6 `Updating Layer Widths`

1: $W(current\_layer) \leftarrow W(current\_layer) - \_n\_width$
2: $W(new\_layer) \leftarrow W(new\_layer) + \_n\_width$
3: **if** $new\_layer(v_i) > current\_layer(v_i)$ **then**
4: **for all** l_j such that $current\_layer(v_i) \leq l_j > new\_layer(v_i)$ **do**
5: $W(l_j) \leftarrow W(l_j) + outdeg(v_i) * \_nd\_width$
6: **end for**
7: **for all** l_j such that $current\_layer(v_i) < l_j \leq new\_layer(v_i)$ **do**
8: $W(l_j) \leftarrow W(l_j) - indeg(v_i) * \_nd\_width$
9: **end for**
10: **else**
11: **for all** l_j such that $current\_layer(v_i) \leq l_j > new\_layer(v_i)$ **do**
12: $W(l_j) \leftarrow W(l_j) + indeg(v_i) * \_nd\_width$
13: **end for**
14: **for all** l_j such that $current\_layer(v_i) > l_j \geq new\_layer(v_i)$ **do**
15: $W(l_j) \leftarrow W(l_j) - outdeg(v_i) * \_nd\_width$
16: **end for**
17: **end if**

The AntColony class

The main method of this 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 layer numbers assigned to vertices are updated.

7 Experiments and Results

Experiments to evaluate the performance of Ant Colony layering algorithm were conducted over a set of 1277 directed graphs (the AT&T graphs) available from http://www.graphdrawing.org.

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 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 nodes count (DNC), 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 on Fig. **??** and Fig. 4. It is visible 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 diagram on Fig. **??**). This is a result of the fact that when 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 less nodes 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 nodes are taken into account (the first diagram on Fig. 4). 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 on Fig. 4). Here clearly the winner is MinWidth followed by the MinWidth with PL and the AntColony both showing close results.

Next criteria used were the height of the layerings and the number of dummy nodes (DNC). The results are shown on Fig. **??** and Fig. **??**. 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 from 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 on Fig. **??**). The Dummy Node Count (DNC) 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. 7 and Fig. 8 the ED of the layrings 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 good to see that the RT of our algorithm is not much higher when LPL and MinWidth are combined with the PL heuristic.

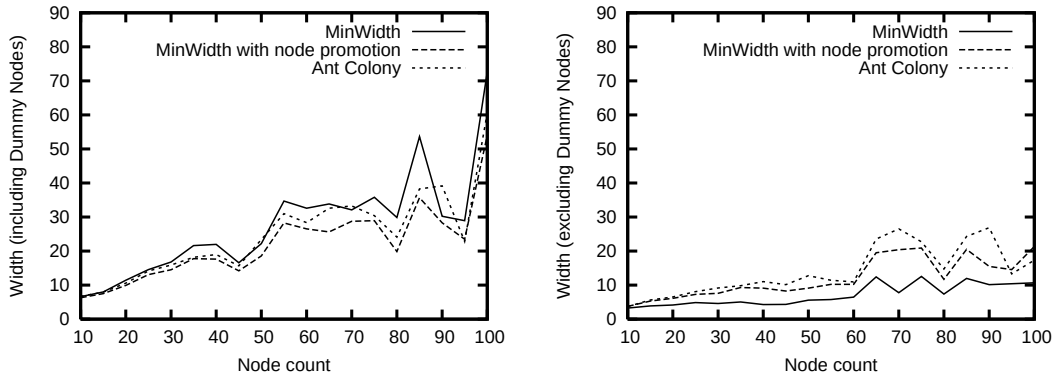

Figure 4: Width of Ant Colony Layering Compared with MinWidth and MinWidth with PL

8 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 $\beta = 5$ 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

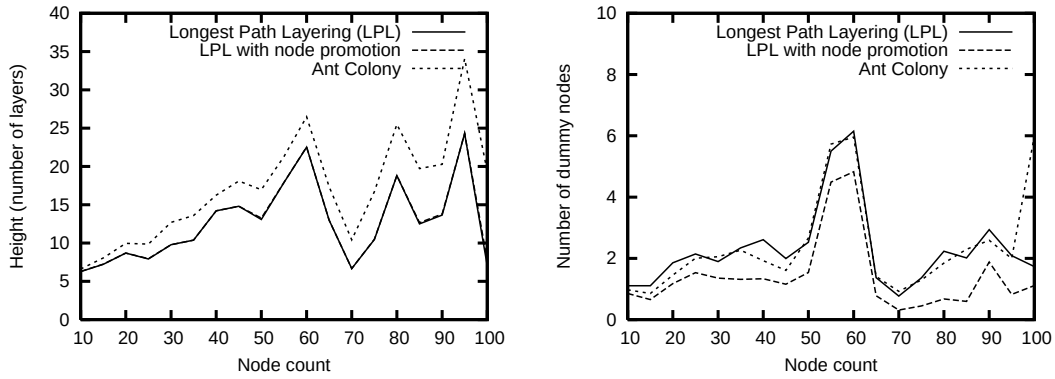

Figure 5: Height and DNC of Ant Colony Layering Compared with LPL and LPL with PL

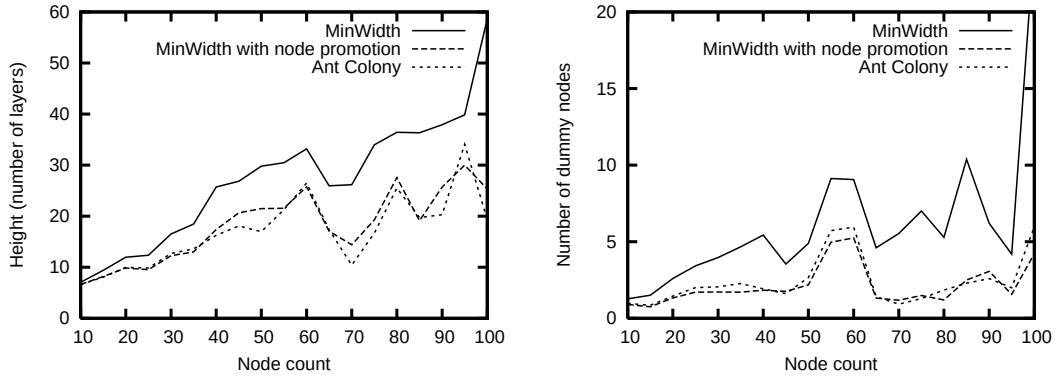

Figure 6: Height and DNC of Ant Colony Layering Compared with MinWidth and MinWidth with PL

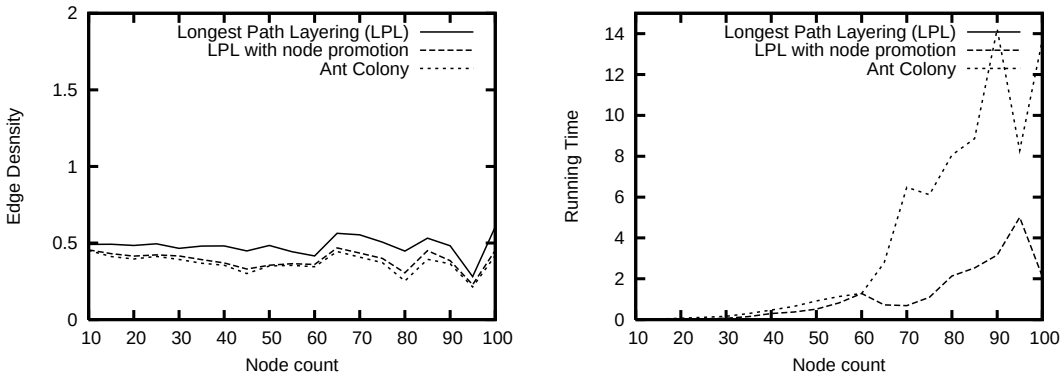

Figure 7: Edge density and Running time of Ant Colony Layering Compared with LPL and LPL with PL

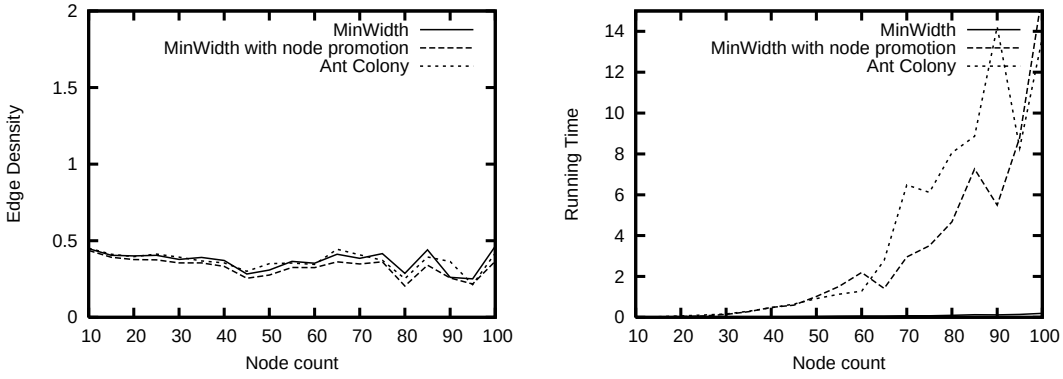

Figure 8: Edge density and Running time of Ant Colony Layering Compared with MinWidth and MinWidth with PL

used as respective values for those two parameters in our further investigations. Another parameter that we have researched 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 nott justify the longer running time and therefore the `_nd_width` = 1 will be used in our experiments.

9 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 (slightly worse) in the key area for each of the two algorithms and in the same time outperforms them in the other layering criteria gives us hope 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 and this is not a surprise because the Ant Colony exploits the LPL to build its starting layering. 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.

10 Further research

In the near future the following directions will be followed.

1. Establish the relation between the number of vertices of the graph being layered and the value (influence) of the heuristic information in 9. In other words if the average number of vertices per layer grows significantly this would cause too steep a decrease of heuristic values. At some point it might turn that they are much smaller than the pheromone values and from there all the decisions will be made almost solely based on the pheromone values. This could lead to a stagnation in the search process forcing ants to follow the same routs on the construction graph.

To detect such a trend (if present) the algorithm will be run on single graph and the final layering will be visualised using LEDA's GraphWin class.

2. Run the algorithm against big graphs. These would be graphs with couple of thousends of vertices. The idea behind this is to harness to power of the Ant Colony and hopefully see its strength when dealing with much larger construction graph than the ones we used in the tests conducted so far.

References

[1] Stigmergy. Definition by Pierre-Paul Grasse. Wikipedia http://en.wikipedia.org/wiki/Stigmergy.

[2] Giusepe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the visualization of graphs*. Prentice Hall, Inc., New Jersey, USA, 1999.

[3] Jürgen Branke, Stefan Leppert, Martin Middendorf, and Peter Eades. Width-restricted layering of acyclic digraphs with consideration of dummy nodes. *Information Processing Letters*, 81(2):59–63, 2002.

[4] E. G. Coffman and R. L. Graham. Optimal scheduling for two processor systems. *Acta Informatica*, 1:200–213, 1972.

[5] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. The Ant System: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics Part B: Cybernetics*, 26(1):29–41, 1996.

[6] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. The MIT Press, Cambridge, Massachusetts and London, England, 2004.

[7] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A technique for drawing directed graphs. *Software Engineering*, 19(3):214–230, 1993.

[8] Patrick Healy and Nikola S. Nikolov. How to layer a directed acyclic graph. In *GD '01: Revised Papers from the 9th International Symposium on Graph Drawing*, pages 16–30, London, UK, 2002. Springer-Verlag.

[9] G. L. Nemhauser and L. A. Wolsey. Integer and combinatorial optimization. *Wiley-Interscience series in discrete mathematics and optimization*, 1988.

[10] Nikola S. Nikolov and Alexandre Tarassov. Graph layering by promotion of nodes. *Discrete Applied Mathematics*, 154:p.848–860, 1 April 2006.

[11] Nikola S. Nikolov, Alexandre Tarassov, and Jürgen Branke. In search for efficient heuristics for minimum-width graph layering with consideration of dummy nodes. *The ACM Journal on Experimental Algorithmics*, 10, 2004.

[12] Kozo Sugiyama. *Graph Drawing and Applications for software and knowledge engineers*, volume 11 of *Series on Software Engineering and Knowledge Engineering*. World Scientific, 2002.

[13] Alexandre Tarassov, Nikola S. Nikolov, and Jürgen Branke. A heuristic for minimum-width graph layering with consideration of dummy nodes. *Lecture Notes in Computer Science*, 3059:570–583, 2004.