An algorithm for Performance Analysis of Single-Source Acyclic graphs

Gabriele Mencagli

September 26, 2011

In this document we face with the problem of exploiting the performance analysis of acyclic graphs of cooperating computation modules. In particular we need an algorithm designed for solving the following problem:

Problem 0.1 (Steady-state analysis of acyclic computation graphs)**.** *Given an acyclic computation graph $G = (V, E)$, in which nodes represent computation modules and edges are data streams, we need a procedure that determines the inter-departure times, that is the effective performance behavior at steady-state, of each node in the graph.*

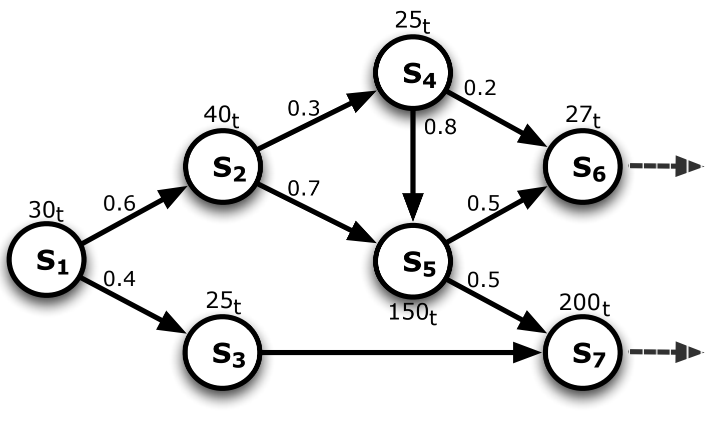


Figure 1: An acyclic computation graph labeled with the ideal service times of each node and the routing probabilities.

In Figure 1 is depicted an example of an acyclic computation graph of seven modules working asynchronously and cooperating by exchanging messages. In the graph each node is labeled with its ideal service time and, when a node has multiple out-going edges, they are labeled with the corresponding probability of transmission. We need an algorithm that has the following features:

- it needs to perform an ordered graph traversal: to correctly establish for each node its inter-arrival time and thus its utilization factor, each node should be visited only when all its in-coming neighbors have been visited and their inter-departure times correctly determined;

- for each node of the graph it is necessary to calculate its inter-arrival time and its utilization factor in order to discover if it is a bottleneck or not. We have two possible situations: (1) the currently visited node is not a bottleneck, i.e. its ideal service time is equal or less than its inter-arrival time and consequently the node influences neither the inter-departure times of the already visited nodes nor of the nodes that are still to be explored; (2) the current node is a bottleneck and it influences the inter-departure times of the previously explored nodes that will be properly incremented.

The first requirement implies that the nodes of the graph should be visited according to a specific ordering. As it is known from basic notions in Graph Theory, every directed acyclic graph has at least one *topological ordering*, i.e. an ordering of its nodes such that the starting vertex of every edge occurs earlier in the ordering than the ending vertex. It can also be shown that an acyclic graph can have multiple admissible topological orderings. Therefore let us suppose to have one of these topological orderings as input of the algorithm. We can observe that if the algorithm visits the nodes following this ordering, the first requirement will be achieved: each node will be visited iff all its in-coming neighbors have already been explored. An example of a topological ordering of the graph in Figure 1 is depicted in Figure 2.

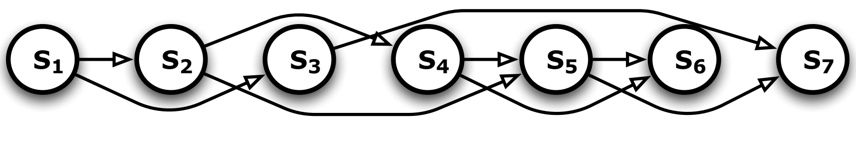


Figure 2: A topological ordering of the graph depicted in Figure 1.

We need a data-structure that represents a general node of the graph. Each node n has four numerical attributes that describe: (1) its inter-arrival time; (2) its ideal service time; (3) its inter-departure time; (4) its utilization factor. Moreover the node maintains a list **OUT** of references to out-going neighbors and a list **IN** of pairs $(n^{'}, p)$, where $n^{'}$ is a reference to one of its in-coming neighbors which transmits to n with probability p.

Function `of the node data-structure`

```
1 Class Node {
2     double $T_A$;
3     double $T_S$;
4     double $T_p$;
5     double $\rho$;
6     ListNode $OUT$;
7     ListPair $IN$;
8 }
```

These data-structures are properly initialized at the beginning of the algorithm execution. For each node the service time variable and the IN and OUT lists are properly initialized according to the structure of the input graph. The inter-arrival, the inter-departure times and the utilization factors will be calculated by the algorithm. For each sink node we assume the presence of a fictitious out-going edge such that the inter-departure time can always be defined. For each source node (although in-coming edges do not exist), the inter-arrival time is kept to be equal to the inter-departure time from that node (and furthermore at the beginning of the execution it coincides with the ideal service time of the node too). The algorithm evolves as follows:

1. The inputs are a directed graph $G = (V, E)$ and one of its topological ordering S (represented as an array of $|V|$ nodes);
2. The algorithm performs the graph traversal by visiting each node following the ordering S. For each explored node its inter-arrival time and its actual utilization factor are determined. Therefore we are able to identify if the node is a bottleneck or not;
3. If the currently explored node is not a bottleneck ($\rho \leq 1$), its inter-departure time equals its inter-arrival time and the graph traversal continues with the next node in the topological ordering S;
4. If the explored node is a bottleneck ($\rho > 1$), its inter-departure time coincides with its ideal service time. At this point the algorithm needs to update the inter-departure times of the nodes already visited in the graph ordering;
5. The algorithm ends when all the nodes have been explored and no bottleneck has been discovered (i.e. nodes have an utilization factor less or equal to 1).

The fourth point is the most critical one. In this section we introduce a formal algorithm for acyclic graphs in which ***there is exactly one source node***. In this case we are able to define an efficient algorithm whose correctness can be proved by introducing the following invariant property:

Invariant 0.2. *When the i-th node in the input topological ordering S is visited, all the previously explored nodes (i.e. from the first one to the $(i-1)$-th of the ordering) have an utilization factor less or at most equal to* 1.

The invariant is satisfied at the beginning of the execution. Every topological ordering of a single source graph starts with the source node. As stated before, for this node its inter-arrival time is initialized to the ideal service time, thus its utilization factor is initially equal to 1. If, during the graph traversal, no bottleneck node is discovered, the algorithm will end when the last node is visited. In this case for each node its inter-departure time equals its inter-arrival time and the graph analysis can trivially be completed.

On the other hand let us suppose that when the i-th node of ordering is visited, its utilization factor is greater than 1. This situation is depicted in Figure 3. We denote the currently discovered bottleneck the node S_b at position i of the topological ordering. Its ideal service time T_b is greater than its actual calculated inter-arrival time T_A (i.e. $\rho_b > 1$). Since, by the invariant, every previous node in the ordering has already been visited and its utilization factor is less (or equal) to 1, the inter-arrival time to S_b can be expressed in function of the actual inter-departure time T_{p_S} from the unique source node of the graph.

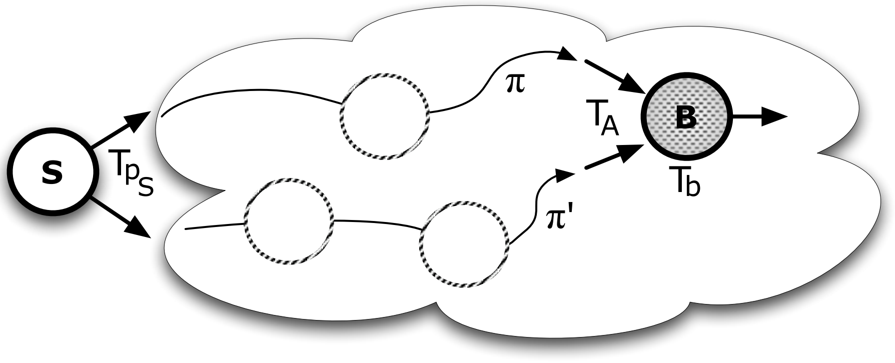


Figure 3: Bottleneck discovery.

To this end we take the set $\mathscr{P}(S_b)$ of all the paths in the graph starting from the source node and ending to the current bottleneck node S_b. A path π is an ordered sequence of edges such that the origin of each is equal to the destination of its predecessor edge. E.g:

$$\pi = \Big\langle (N_1, p_1, N_2), (N_2, p_2, N_3), \ldots, (N_{k-1}, p_{k-1}, N_k) \Big\rangle$$

Where a directed edge is represented as a triple $e = (N, p, N^{'})$ where the first and the third element are the two end-point vertices of the edge and the second element is the probability that the first node transmits to the second one. For brevity we indicate with $e.p$ the probability corresponding to the edge e. By invariant all the nodes preceding S_b in the ordering have $\rho \leq 1$, thus we can determine the inter-arrival time to S_b by taking all the paths starting from the source node and ending to S_b, and iteratively applying the *Server Partitioning theorem* to calculate the inter-arrival time to the bottleneck:

$$T_A = \left(\sum_{\forall \pi \in \mathscr{P}(S_b)} \left(\frac{\prod_{\forall e \in \pi} e.p}{T_{p_S}} \right) \right)^{-1} \tag{1}$$

As we have seen the presence of the new discovered bottleneck node S_b influences the inter-departure times of all the previously explored nodes: i.e. they must be properly corrected. After this correction, the new inter-arrival time $T_A^{'}$ to S_b must be equal to its ideal service time T_b. Thus, similarly to the previous case, we can express the new inter-arrival time to S_b in function of the corrected inter-departure time from the source node $T_{p_s}^{'}$:

$$T_A^{'} = \left(\sum_{\forall \pi \in \mathscr{P}(S_b)} \left(\frac{\prod_{\forall e \in \pi} e.p}{T_{p_S}^{'}} \right) \right)^{-1} = T_b \tag{2}$$

In order to understand how we can correct the inter-departure time from the source, we can express the following relation: $T_{p_s}^{'} = T_{p_s} \cdot \alpha$ where α is a multiplicative factor. At this point we need to find an α such that:

$$\left(\sum_{\forall \pi \in \mathscr{P}(S_b)} \left(\frac{\prod_{\forall e \in \pi} e.p}{\alpha T_{p_S}} \right) \right)^{-1} = T_b$$

From which we obtain the right expression for the coefficient α:

$$\left(\sum_{\forall \pi \in \mathscr{P}(S_b)} \left(\frac{\prod_{\forall e \in \pi} e.p}{\alpha T_{p_S}} \right) \right)^{-1} = \frac{T_{pS}}{\sum_{\forall \pi \in \mathscr{P}(S_b)} \left(\prod_{\forall e \in \pi} e.p \right)} = \frac{T_b}{\alpha}$$

We can observe that the first element of the equation is the original inter-arrival time T_A, thus we can write:

$$T_A = \frac{T_b}{\alpha} \Longrightarrow \alpha = \frac{T_b}{T_A} = \rho_b$$

Therefore, when a new bottleneck node is discovered, we can correct the inter-departure time from the source node by multiplying the old inter-departure time by the utilization factor of the bottleneck node that has been discovered.

Proposition 0.3 (Invariant preservation)**.** *During the algorithm execution, if the i-th node of the topological ordering is a bottleneck ($\rho_i > 1$), we need to correct the inter-departure time from the source by multiplying this value by the utilization factor ρ_i. Then the algorithm is re-started from the beginning and, this time, when the i-th node is reached its utilization factor will be equal to* 1 *and all the previous nodes will continue to have an utilization factor less than* 1.

Proof. This proposition proves the correctness of the algorithm. Multiplying the inter-departure time of the source by the utilization factor of the discovered bottleneck is the only way to achieve a new corrected inter-arrival time $T_A^{'}$ to S_b equals its service time T_b. Since $\rho_b > 1$ this means that the corrected inter-departure time $T_{p_s}^{'}$ is greater than the original one T_{p_s}, and thus the nodes preceding S_b in the ordering will continue to have an utilization factor less than 1. □

1 Algorithm description

Algorithm 2 presents an automatic procedure for single source acyclic graph analysis. The algorithm proceeds in the following fashion. All the nodes are visited according to an input topological ordering. For each node is calculated its inter-arrival time by accessing its IN neighbor list (row 4). After that the utilization factor of the node is determined (row 5) and the bottleneck and non-bottleneck cases are examined. The most simply situation is when no bottleneck is discovered: in this case (from row 9 to 11) the inter-departure time of the current node is equal to the calculated inter-arrival time. Otherwise, if a bottleneck is discovered (from row 6 to 8), the inter-departure time of the source (first node in the ordering) is corrected as we have said and the visit re-starts from the beginning.

Algorithm 2: Steady-state Analysis(G, S)

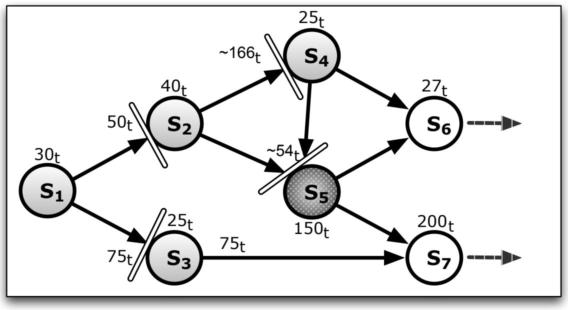
```
Data: a single-source acyclic graph $G = (V, E)$ and one of its topological
      ordering $S$.
Result: at the end of the execution the attribute $T_p$ of each node
        corresponds to its inter-departure time at steady-state.
1  begin
2      i ← 1;
3      while $i \leq |V|$ do
4          $S[i].T_A = \left( \sum_{(u,p) \in S[i].IN} \frac{p}{u.T_p} \right)^{-1}$;
5          $S[i].\rho = \frac{S[i].T_S}{S[i].T_A}$;
6          if $S[i].\rho > 1$ then bottleneck case
7              $S[1].T_p = S[1].T_p \times S[i].\rho$;
8              i ← 1;
9          else not bottleneck case
10             $S[i].T_p = S[i].T_A$;
11             i ← $i + 1$;
12
13 end
```

Proposition 1.1 (Time complexity of steady-state analysis)**.** *At the worst case the time complexity of steady-state analysis is $O(|V|^2)$ for sparse graphs and $O(|V|^3)$ for dense graphs.*

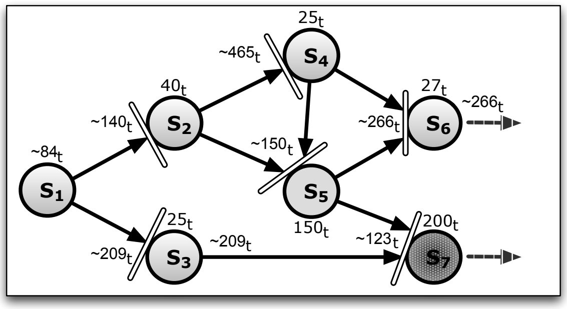
Proof. The cost in terms of time complexity of a graph traversal (without any restart) is $O(|V| + |E|)$, since for each node its list IN is visited once (see row 4). The traversal of the graph needs to be re-started whenever a bottleneck node is discovered. Let us consider B the number of bottleneck nodes that are discovered during the algorithm execution, where $0 \leq B \leq |V|$. The complexity of steady-state analysis is $O\left(B \cdot (|V| + |E|)\right)$ where at the worst case $B = |V|$ (i.e. whenever a node is explored for the first time it results a bottleneck). Therefore for sparse graphs (where $|E| = O(|V|)$) the time complexity is $O(|V|^2)$ whereas for dense graphs (where $|E| = O(|V|^2)$) is $O(|V|^3)$. We can also notice that if no bottleneck node is discovered (i.e. $B = 0$), the time complexity of the algorithm is the same of a simple graph traversal. □

For completeness we can observe that the algorithm needs as inputs a topological ordering S. As it is well-known the time complexity for finding a topological ordering is the same of a DFS (*depth-first search*) traversal of the graph, i.e. $O(|V| + |E|)$. Thus the cost of Algorithm 2 dominates the overall time complexity for the steady-state analysis.

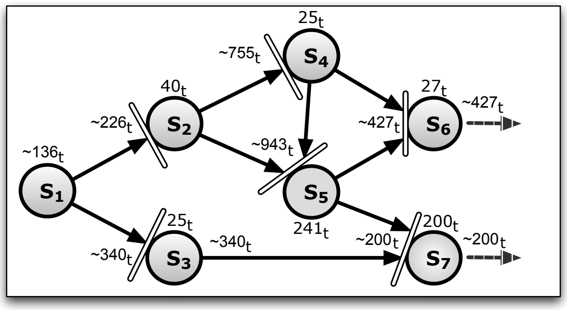
Example. In Figure 4 is provided an example of steady-state analysis of an acyclic graph. Let us consider the input graph depicted in Figure 1 labeled with the ideal service times of each node and the routing probabilities. Figure 4 depicts the different phases of the algorithm execution following the topological ordering shown in Figure 2. Gray nodes represent explored vertices, white nodes



(a) The graph traversal starts from the node S_1. It is the unique source so its inter-arrival time is initially equal to its service time. Next, node S_2 is explored: the node is not a bottleneck since its inter-arrival time ($50t$) is greater than its service time. The same thing happens for nodes S_3 (with inter-arrival time $75t$) and for S_4 with inter-arrival time $166t$. When S_5 is discovered, its is a **bottleneck**: its inter-arrival time $54t$ is less than its service time $150t$ and its utilization factor is $\rho_5 = 2.79$. Therefore we update the inter-departure time of the source node that passes from $30t$ to $30t \cdot \rho_5 = 84t$.



(b) At this point the graph traversal re-starts from node 1. When S_5 is reached, it is not a bottleneck anymore (i.e. $\rho_5 = 1$). Now the node S_6 is explored and it is not a bottleneck (its inter-arrival time is $266t$). Then the last node S_7 is visited and its inter-arrival time $123t$ is less than its service time $200t$. So this node is a **bottleneck** and its utilization factor is $\rho_7 = 1.62$. Hence we update the inter-departure time of the source node that passes from $84t$ to $84t \cdot \rho_7 = 136t$.



(c) The graph traversal re-starts from node 1. At this point no bottleneck node is identified: i.e. for every node in the graph its utilization factor is now lower or equal to 1. The algorithm terminates correctly providing the steady-state behavior of the acyclic graph.

Figure 4: An example of steady-state analysis of an acyclic graph.

Node	**Service time**	**Inter-departure time**	**Utilization factor**
S_1	$30t$	$136t$	.2205
S_2	$40t$	$226t$	.1770
S_3	$25t$	$340t$	.0736
S_4	$25t$	$755t$	.0326
S_5	$150t$	$241t$	.6224
S_6	$27t$	$427t$	.0633
S_7	$200t$	$200t$	1

Table 1: Results of steady-state analysis.

correspond to vertices that are still to be explored whereas a point-based black node is the currently discovered bottleneck. The final results are shown in Table 1.

2 Impact of randomness on Single-Source graphs Analysis

We conclude the analysis of acyclic computation graphs by providing a brief discussion about the impact of the randomness on the accuracy of the results achieved with the algorithm. In the previous sections we have assumed deterministic service times for each node of the graph: i.e. for each node its ideal service time is a fixed constant value. In this case the accuracy of the algorithm has been evaluated on several example graphs through a Queueing Network simulator (*Java Modelling Tool*). The simulation results demonstrate an absolute precision of the algorithm which is able to quantify the steady-state behavior of each node.

Things can become different if we introduce randomness, i.e. if we suppose stochastic random variables that model the service time of each node. Here, the service time assumes stochastic values following a probability density function with a known average value. A valuable modeling consists in assuming that the service processes of each node follow an *exponential distribution*. With this assumption each node in the network is modeled as a M/M/1 queue (instead of D/D/1 queues as in the previous discussion). The main property of this distribution is the *memoryless*: if we assume that each service request (task) is independent from the others, the service time for completing a task does not depend on the service times spent for the previous tasks calculated by the node. For stream-based parallel computations the exponential approximation is usually an acceptable modeling approach.

All the basic results for graph analysis, i.e. the inter-departure, server partitioning and multiple clients theorems are still valid if we assume exponential random variables modeling the ideal service times of each node in the graph. On the other hand, compared to the deterministic case, in the exponential case the size of each queue plays an important role for attenuating the randomness impact. In fact we expect that the results of the steady-state analysis approximates well the behavior of a M/M/1 network if, for each node, the queue size is large enough (but still bounded). For this reason we have simulated the behavior of the network shown in Figure 1, in which ideal service times are assumed to be the average values of corresponding exponential random variables. The simulations have been performed by using Java Modelling Tools and by varying the size of each queue. Tests for 25, 6, 3, 2 and 1 buffer positions are depicted in Table 2.

Node	**Er %.(25)**	**Er %.(6)**	**Er %.(3)**	**Er %.(2)**	**Er %.(1)**
S_1	.5014	2.4086	7.1857	10.2389	18.0247
S_2	.7897	2.4254	7.1375	10.3436	18.3095
S_3	.6359	2.8383	6.9518	10.1564	17.6470
S_4	.3411	1.8848	6.8147	9.46308	18.2592
S_5	.7325	2.2014	6.6678	9.77189	17.8800
S_6	.1996	2.6694	6.8376	9.64912	17.3708
S_7	.0820	.9448	6.4509	9.43553	17.6844

Table 2: Accuracy of steady-state analysis in function of the queue size of each node.

In the table are reported for each queue node the percentage errors between the simulation results and the inter-departure times obtained by the algorithm execution. As we can expect if we decrease the size of each queue the errors increase. For this example we can observe that for large enough queue size (up to 12 buffer positions), the errors are less than 1% for each node. For very limited queue size (i.e. 3, 2 and 1 positions), the errors are less than 8, 11 and 19% for each node.