ANT COLONY SYSTEMS FOR LARGE SEQUENTIAL ORDERING PROBLEMS

R. Montemanni a,c, D.H. Smith b, L.M. Gambardella a

a Istituto Dalle Molle di Studi sull’Intelligenza Artificiale (IDSIA)
Galleria 2, CH-6928 Manno-Lugano, Switzerland
{roberto, luca}@idsia.ch

b Division of Mathematics and Statistics, University of Glamorgan
Pontypridd CF37 1DL, United Kingdom
dhsmith@glam.ac.uk

ABSTRACT

The sequential ordering problem is a version of the asymmetric traveling salesman problem where precedence constraints on vertices are imposed. A tour is feasible if these constraints are respected, and the objective is to find a feasible solution with minimum cost.

The sequential ordering problem models a lot of real world applications, mainly in the fields of transportation and production planning.

In this paper we propose an extension of a well known ant colony system for the problem, aiming at making the approach more efficient on large problems. The extension is based on a problem manipulation technique that heuristically reduces the search space.

Computational results, where the extended ant colony system is compared to the original one, are presented.

KEYWORDS: Problem manipulation techniques, ant colony optimization, asymmetric traveling salesman, scheduling.

1. INTRODUCTION

The Sequential Ordering Problem (SOP), also referred to as Asymmetric Traveling Salesman Problem with Precedence Constraints, can be modeled in graph theoretical terms as follows. A complete directed graph \( D = (V, A) \), where \( V \) is the set of nodes and \( A = \{ (i,j) | i, j \in V \} \) is the set of arcs, is given. A cost \( c_{ij} \in \mathbb{N} \) is associated with each arc \( (i,j) \in A \). Without loss of generality we assume that a fixed starting node \( 1 \in V \) is given. It has to precede all the other nodes. The tour is also closed at node 1, after all the other nodes have been visited. This to create an analogy with the asymmetric traveling salesman problem \( c_{11} = 0 \ \forall i \in V \).

Furthermore we are given an additional precedence digraph \( P = (V, R) \), defined on the same node set \( V \) as \( D \). An arc \( (i,j) \in R \), represents a precedence relationship, i.e. \( i \) has to precede \( j \) in every feasible tour. We will denote such a relation as \( i \prec j \) in the remainder of the paper. The precedence digraph \( P \) must be acyclic in order for a feasible solution to exist. We also assume it is transitively closed, since \( i \prec k \) can be inferred from \( i \prec j \) and \( j \prec k \). Notice that for the last arc traversed by a tour (entering node 1), precedence constraints do not apply. A tour that satisfies precedence relationships is called feasible. The objective of the SOP is to find a feasible tour with the minimal total cost.

It is interesting to observe that SOP reduces to the classical asymmetric traveling salesman problem (ATSP) in the case where no precedence constraint is given. This observation implies that SOP is \( \mathcal{NP} \)-hard, being a generalization of the ATSP.

The SOP models real-world problems such as production planning (Escudero [7]), single vehicle routing problems with pick-up and delivery constraints (Pulleyblank and Timlin [13], Savelsbergh [14]) and transportation problems in flexible manufacturing systems (Ascheuer [1]).

Sequential ordering problems were initially solved as constrained versions of the ATSP, especially for the development of exact algorithms. The main effort has been put into extending the mathematical definition of the ATSP by introducing new classes of valid inequalities to model the additional constraints. The first mathematical model for the SOP was introduced in Ascheuer et al. [2] where a cutting plane approach was proposed to compute lower bounds on the optimal solution. In Escudero et al [8], a Lagrangean relaxation method was described and embedded into a branch and cut algorithm. Ascheuer [1] has proposed a new class of valid inequalities and has described a new
branch-and-cut method for a broad class of SOP instances. This is based on the polyhedral investigation carried out on ATSP problems with precedence constraints by Balas et al. [3]. The approach in [1] also investigates the possibility to compute and improve sub-optimal feasible solutions starting from the upper bound provided by the polyhedral investigation. The upper bound is the initial solution of a heuristic phase based on well-known ATSP heuristics that are iteratively applied in order to improve feasible solutions. These heuristics do not handle constraints directly: infeasible solutions are simply rejected. A branch and bound algorithm with lower bounds obtained from homomorphic abstractions of the original search space has been presented in Hernádvölgyi [11] (see also [12]). A genetic algorithm has been described in Guerriero and Mancini [10]. Gambardella and Dorigo [9] presented an approach based on Ant Colony Optimization enriched with sophisticated local search procedures. This last method can be classified as state-of-the-art for the sequential ordering problem and will be described in detail in Section 2.

The contribution of the present article will be an extension of the method described in [9], aiming at improving the performance of the method on large problems. The novel idea at the basis of the extension is a problem manipulation technique.

2. ANT COLONY OPTIMIZATION

The Ant Colony System (ACS) algorithm is an element of the Ant Colony Optimization (ACO) family of methods (Dorigo et al. [5]). These algorithms are based on a computational paradigm inspired by real ant colonies and the way they function. The main underlying idea was to use several constructive computational agents (simulating real ants). A dynamic memory structure, which incorporates information on the effectiveness of previous choices, based on the obtained results, guides the construction process of each agent. The behavior of each single agent is therefore inspired by the behavior of real ants.

The paradigm is based on the observation, made by ethologists, that the medium used by ants to communicate information regarding shortest paths to food, consists of pheromone trails. A moving ant lays some pheromone on the ground, thus making a path by a trail of this substance. While an isolated ant moves practically at random (exploitation), an ant encountering a previously laid trail can detect it and decide, with high probability, to follow it, thus reinforcing the trail with its own pheromone (exploitation). What emerges is a form of autocatalytic process where the more the ants follow a trail, the more attractive that trail becomes to be followed. The process is thus characterized by a positive feedback loop, where the probability with which an ant chooses a path increases with the number of ants that previously chose the same path. The mechanism above is the inspiration for the algorithms of the ACO family.

2.1. Ant Colony Optimization for the SOP

As said in the previous section, application of an ACO algorithm to a combinatorial optimization problem requires definition of a constructive algorithm and possibly a local search. Accordingly, a constructive algorithm called ACS-SOP in which a set of artificial ants builds feasible solutions to the SOP has been designed, together with a local search specialized for the SOP that takes these solutions to their local optimum. The resulting algorithm is a Hybrid Ant System for the SOP called HAS-SOP, which is described in detail in Gambardella and Dorigo [9].

2.1.1. Construction phase (ACS-SOP)

ACS-SOP is strongly based on the Ant Colony System algorithm (Dorigo and Gambardella [6]). ACS-SOP implements the constructive phase of HAS-SOP, and its goal is to build feasible solutions for the SOP. It generates feasible solutions with a computational cost of order $O(|V|^2)$.

Informally, ACS-SOP works as follows. Ants are sent out sequentially (not in parallel). Each ant iteratively starts from node 1 and adds new nodes until all nodes have been visited. When in node $i$, an ant applies a so-called transition rule, that is, it probabilistically chooses the next node $j$ from the set $F(i)$ of feasible nodes. $F(i)$ contains all the nodes $j$ still to be visited and that all nodes that have to precede $j$, according to precedence constraints, have already been inserted in the sequence.

The ant in node $i$ chooses the next node $j$ to visit on the basis of two factors: the heuristic desirability $\eta_{ij}$ here defined as $1/c_{ij}$, and the pheromone trail $\tau_{ij}$, that contains a measure of how good it has been in the past to include arc $(i, j)$ into a solution (it is the “memory” of the colony). The next node to visit is chosen with probability $q_0$ as the node $j, j \in F(i)$, for which the product $\tau_{ij} \cdot \eta_{ij}$ is highest (deterministic rule), while with probability $1 - q_0$ the node $j$ is chosen with a probability given by

$$p_{ij} = \frac{\tau_{ij} \cdot \eta_{ij}}{\sum_{\tilde{j} \in F(i)} (\tau_{i\tilde{j}} \cdot \eta_{i\tilde{j}})}$$

(i.e., nodes connected by arcs with higher values of $\tau_{ij} \cdot \eta_{ij}, j \in F(i)$, have higher probability of being chosen).
The value $q_0$ is given by $q_0 = 1 - s/|V|$. The parameter $s$ represents the number of nodes we would like to choose using the probabilistic transition rule, independently of the number of nodes of the problem.

In ACS-SOP only the best ant, that is the ant that built the shortest tour since the beginning of the computation, is allowed to deposit pheromone trail. The rationale is that in this way a “preferred route” is memorized in the pheromone trail matrix, and future ants will use this information to generate new solutions in a neighbourhood of this preferred route. If we refer to the shortest path generated since the beginning of the computation as $OptPath_{Best}$, and to its cost as $L_{Best}$, $\forall \{(i,j) \in OptPath_{Best}\}$, we have the following formula for pheromone update:

$$\tau_{ij} = (1 - \rho) \cdot \tau_{ij} + \frac{\rho}{L_{best}} \quad (1)$$

Pheromone is also updated during solution building. In this case, however, it is removed from visited arcs. In other words, each ant, when moving from node $i$ to node $j$, applies a pheromone updating rule that causes the amount of pheromone trail on arc $(i,j)$ to decrease.

The rule is:

$$\tau_{ij} = (1 - \psi) \cdot \tau_{ij} + \psi \cdot \tau_0 \quad (2)$$

where $\tau_0$ is the initial value of trails. It was found that good values for the algorithm’s parameters are $\tau_0 = (FirstSolution \cdot n)^{-1}$, $\rho = \psi = 0.1$, $s = 10$, where FirstSolution is the length of the shortest solution generated by the ant colony following the ACS-SOP algorithm without using the pheromone trails. Experience has shown these values to be robust. The number of ants in the population was set to 10. The rationale for using formula (2) is that it causes ants to eat away pheromone trail while they build solutions so that a certain variety in generated solutions is assured (if pheromone trail was not consumed by ants, they would tend to generate very similar tours).

2.1.2. Complete algorithm (HAS-SOP)

The HAS-SOP algorithm is the ACS-SOP algorithm augmented by local search. Local search is an optional component of ACO algorithms, although it has been shown since early implementations that it can greatly improve the overall performance of the ACO metaheuristic when static combinatorial optimization problems are considered (Dorigo and Gambardella [6]).

In HAS-SOP, local search is applied once each ant has built its solution: the solution is carried to its local optimum by an application of the extremely efficient SOP-3-exchange local search routine. This local search routine is a specialization to the sequential ordering problem of a known local search method for the asymmetric traveling salesman problem (Savelsbergh [14]). It is able to directly handle multiple constraints without increasing the computational complexity of the original local search. Since the description of such a local search method is out of the scope of this paper (although the local search routine is used by the algorithm we propose) we refer the interested reader to Gambardella and Dorigo [9] for its detailed description. Locally optimal solutions are then used to update pheromone trails on arcs, according to the pheromone trail update rule (1).

The algorithm stops when a fixed CPU time has elapsed.

3. ARTIFICIAL PRECEDENCE CONSTRAINTS

It is trivial to observe that adding precedence constraints to a given problem reduces its search space, making the problem potentially easier to solve. Starting from this observation, we developed the method described in the remainder of this section.

The underlying idea is to monitor the solutions generated by HAS-SOP, and to identify precedence patterns common to solutions with a low cost. Once such precedence patterns are identified, they can be added to the original problem as artificial precedence constraints. The resulting problem is likely to be easier than the original one, as it has a reduced solution space. Of course such a heuristic method may cut out all the optimal solutions of the original problem, leading to suboptimal solutions even in the case that the best solution of the modified problem is retrieved.

Formally, the methodology we propose is integrated into the HAS-SOP method and makes use of an additional set of variables $m$. Variable $m_{ij}$ will be an indicator for the “quality” of the solutions in which node $i$ is visited before node $j$. We also need the following additional parameters:

$u$ : number of solutions generated (ants sent out) before the first artificial precedence constraints are added to the problem;

$v$ : number of solutions generated (ants sent out) between two consecutive creations of artificial precedence constraints;

$w$ : (approximate) number of artificial precedence constraints added each time group of new artificial precedence constraints is generated;

The method we propose is integrated into the classical HAS-SOP algorithm as follows. We initialize $m_{ij} = 0 \ \forall (i,j) \in A$.

Each time a new solution $OptPath_k$, with cost $L_k$, is generated by an ant of the colony (and taken down to its local optimum), matrix $m$ is updated as follows:

$$m_{ij} = m_{ij} + \frac{L_1}{L_k} \ \forall i,j \in V, \pi_k(i) < \pi_k(j) \leq \pi_k(i) + z \quad (3)$$
\[ m_{ji} = m_{ji} - \frac{L_i}{L_k} \forall i, j \in V, \pi_k(i) < \pi_k(j) \leq \pi_k(i) + z \] (4)

where \( L_i \) is the cost of the solution generated by the very first ant and \( \pi_k(i) \) is the index of the position occupied by node \( i \) in solution \( OptPath_k \). \( z \) is not a listed parameter (see next page). Its value regulates the width of the window considered for updates.

The first update (equation (3)) reinforces the entry corresponding to a sequence which is in solution \( OptPath_k \). The update is proportional to the inverse of the cost of the solution itself. Equation (4) decreases the value on arcs that are traversed in the opposite direction in the current solution. This second update has been inserted to make those pairs of nodes that do not seem to have a clear ordering relationship less attractive. Notice that only pairs with a positive entry in matrix \( m \) will be potentially transformed into artificial precedence constraints.

We need now to briefly comment on \( z \). Values of \( z \) that are too small might lead to a method where only arcs common to many solutions are identified, and not pairs of nodes that are in the same order (but not necessarily contiguous) in many solutions, as we would like. On the other hand, values of \( z \) that are too large might make the method too sensitive, and lead to a large number of negative entries in matrix \( m \). Notwithstanding these considerations, we decided not to list \( z \) among the parameters because preliminary results suggest that the method is not sensitive at all to changes to (reasonable values of) \( z \). In particular, values within the interval \([4, 10]\) seem to guarantee the best performance. In our method we set \( z = 5 \).

After the first \( u \) solutions are created by ants (and taken down to their local optimum), a first set of (approximately) \( w \) artificial precedence constraints are added to matrix \( P \). The new constraints are selected as the ones not yet present in the precedence digraph \( P \) with the highest entries in matrix \( m \), plus those implied by them by transitivity. If there are less than \( w \) entries of \( m \) with a positive value, then only the precedence constraints corresponding to them will be added to \( P \) (together with those implied by transitivity). The next set of artificial constraints will be added every time \( v \) new solutions have been generated, following the same logic.

The new hybrid ant colony system that makes use of Artificial Precedence Constraints will be referred to as HAS-SOP\(_{APC}\). It is summarized by the pseudo-code of Figure 1.

After some preliminary tests, we identified two prominent (and promising) settings of the parameters of HAS-SOP\(_{APC}\): These settings suggest the following two algorithms arising from method HAS-SOP\(_{APC}\):

- **HAS-SOP\(_{APC}\)^P**: parameters are set as follows: \( u = 20, v = \infty, w = 10 \). This parameter setting leads to the configuration that we will refer to as method HAS-SOP\(_{APC}\)^P, where P stands for preprocessing.

The artificial precedence constraints are added all at once at the beginning of the execution of the conventional HAS-SOP algorithms, which then runs on a steady problem, that has more precedence constraints than the original one.

In detail, 20 solutions (i.e. 2 for each ant) are generated (and taken down to their local optima). The 10 most promising artificial constraints (according to the values in matrix \( m \)) are added to the precedence digraph \( P \) (together with those implied by transitivity). The HAS-SOP algorithm runs on the modified problem for the available computational time remaining.

- **HAS-SOP\(_{APC}\)^C**: Parameters are set up as follows: \( u = 100, v = 1000, w = 1 \). This parameter setting leads to the configuration that we will identify as HAS-SOP\(_{APC}\)^C in the remainder of the paper, where \( C \) stands for cumulative. In this case artificial precedence constraints are added in a cumulative fashion during the whole execution of the conventional HAS-SOP algorithm. In particular, every time 1000 new solutions are generated, matrix \( m \) is examined and the precedence constraints associated with the entry of the matrix with the highest positive value (if any) is added to the precedence digraph \( P \) (together with constraints implied by transitivity). The precedence digraph \( P \) evolves therefore during the whole computation, getting more and more restrictive.

The two methods listed above will be considered for the computational experiments described in Section 4.

### 4. COMPUTATIONAL RESULTS

The aim of this section is to compare the original HAS-SOP algorithm with the modified methods HAS-SOP\(_{APC}\)^P and HAS-SOP\(_{APC}\)^C, described in Section 3.

All the methods have been coded in C++ (starting from the original implementation of HAS-SOP, see [9]) and all the experiments have been run on a Intel Pentium 4 1.5GHz / 256MB machine. The maximum computation time was set to 600 seconds for all the problems. This computation time should be long enough to let all the methods reach a steady state, where further improvements are unlikely to be found.

#### 4.1. Benchmark problems

The benchmark problems available at TSPLIB\(^3\) have been initially used for testing the new algorithms we propose.

---

\(^3\)http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/.
Unfortunately it was impossible to observe any significant difference in performance between HAS-SOP and HAS-SOP\textsubscript{APC} methods, since the problems tend to be rather easy for modern heuristics (for most of the problems the best solutions have been proven to be optimal, and for the remaining ones no improvement has been registered in the last ten years, and very good lower bounds are available). For this reason, we decided to generate new random problems, bigger and harder to solve than those contained in the (dated) TSPLIB.

The problems we generated, that are publicly available\textsuperscript{2}, were named \(n\text{-r-p}\), where the meaning of each element is as follows:

\[
\begin{align*}
n & : \text{ the number of nodes of the problem, i.e. } V = \{1, 2, \ldots, n\}; \\
r & : \text{ the cost range, i.e. } 0 \leq c_{ij} \leq r \; \forall i, j \in V; \\
p & : \text{ the approximate percentage of precedence constraints, i.e. the number of precedence constraints of the problem will be about } p \leq \frac{n(n-1)}{2}.
\end{align*}
\]

We considered the following values for the parameters above, generating problems for all the possible combinations of them:

- \(n \in \{200, 300, 400, 500, 600, 700\}\);
- \(r \in \{100, 1000\}\);
- \(p \in \{1, 15, 30, 60\}\).

The resulting set of problems covers a wide range of situations, with problems of different size, with different granularity for costs, and with radically different percentages of precedence constraints. The set should provide a good testbed for modern SOP heuristic algorithms.

4.2. Experiments

Five runs are considered for each possible problem/method combination. The results of the experiments are reported in Table 1. The first three columns are parameters of the problems, while the remaining columns are devoted to the presentation of the average and best results obtained by the methods considered. Percentage improvements over the standard HAS-SOP method are reported for HAS-SOP\textsuperscript{P\textsubscript{APC}} and HAS-SOP\textsuperscript{C\textsubscript{APC}} (both for average and best cases). Extra lines have finally been inserted into the table to present averages for percentage improvements on different subsets of the testbed.

Table 1 confirms the intuition that adding artificial precedence constraints, together with the associated search-space reduction, can help ant colony system methods for the sequential ordering problem.

A deeper analysis of the results reported in Table 1 leads to the observation that method HAS-SOP\textsuperscript{P\textsubscript{APC}} works better than HAS-SOP\textsuperscript{C\textsubscript{APC}}. This result, which might be somehow surprising, can be explained by observing that method HAS-SOP\textsuperscript{C\textsubscript{APC}}, contrary to what happens for the simpler method HAS-SOP\textsuperscript{P\textsubscript{APC}}, seems to have two main drawbacks (revealed by other experiments not reported here). In some situations (typically when just a few constraints are specified in the original problem) solutions with very different characteristics are likely to be retrieved by the HAS-SOP method, leading to many negative entries in matrix \(m\). In this case the algorithm is able to add just a few artificial constraints. On the other hand, for problems with many (original) precedence constraints, the strategy of adding constraints during the whole computation tend to quickly overrestrain the search space around a small set of solutions, preventing the algorithm from exploring other (possibly promising) areas of the search-space.

Another interesting phenomenon emerging from Table 1 is that method HAS-SOP\textsuperscript{P\textsubscript{APC}} very rarely obtains worse results than the classic HAS-SOP algorithm. This observation suggests that it is convenient to use HAS-SOP\textsuperscript{P\textsubscript{APC}} because even if it does not provide improvements, it does not “hide” good solutions that the classical method would be able to retrieve.

A further analysis of Table 1 suggests that the improved methods work better for the largest problems, for which the standard method is likely to have more difficulties. This proves once again that artificial precedence constraints help in making the problem easier to handle. Another confirmation of this intuition comes from the observation that the improvements guaranteed by the new methods tend to vanish when \(p\) (percentage of precedence constraints in the original problem) increases, and consequently the search-space is already small and there is not an obvious convenience in reducing it further.

5. CONCLUSIONS AND FUTURE WORK

A problem manipulation method, which creates and adds artificial precedence constraints to the original problem, has been embedded into a well-known ant colony system for the sequential ordering problem.

The extended method induces small improvements in the performance of the classical ant colony system, leading to better results, especially on large and difficult problems.

It is important to observe that the problem manipulation method we propose can be adapted to many other combinatorial optimization methods, and that it is not applicable to ant colony systems only. In our future research we will then try to generalize the manipulation method. We will apply it

\textsuperscript{2}http://www.idsia.ch/~roberto/SOPLIB06.zip.
to different problems, with different algorithms driving the optimization.

Another stream of research will be dedicated to the development of improved versions of $\text{HAS-SOP}^C_{\text{APC}}$, in which artificial constraints can be not only added, but also retracted during the computation.

6. ACKNOWLEDGEMENTS

The authors would like to thank Diego Frei and Filip Klisic for their contributions in the implementation of the new extensions into the original $\text{HAS-SOP}$ code.

7. REFERENCES


1. For each pair \((r, s)\)
   \(\tau_{rs} := \tau_0\)
   \(m_{rs} := 0\)

   EndFor

   # counter := 1

2. For \(k := 1\) to \(m\) do
   Let \(r_k\) be the node where ant \(k\) is located
   \(r_k := 1 /* All ants start from node 1 */\)

   EndFor

   /* The path of agent \(k\) is stored in \(Path_k\) */

   For \(k := 1\) to \(m\) do
     For \(i := 1\) to \(n - 1\) do
       Starting from \(r_k\) compute the set \(F(r_k)\) of feasible nodes
       /* \(F(r_k)\) contains all the nodes \(j\) still to be visited and such that all
       nodes that have to precede \(j\) have already been inserted in the sequence */
       Choose the next node \(s_k\) according to the transition rule (see Section 2.1.1)
       \(Path_k := (r_k, s_k)\)
       \(\tau_{r_k s_k} := (1 - \psi) \cdot \tau_{rs} + \psi \cdot \tau_0 /* This is equation (2) */\)
       \(r_k := s_k\)
     EndFor
    EndFor

    OptPath_k := LocalSearchRoutine(Path_k)

    Compute \(L_k /* L_k is the length of the path OptPath_k */\)
    # For each nodes \(r, s \in V\) such that \(\pi_k(i) < \pi_k(j) \leq \pi_k(i) + 5\)
    # /* \(\pi_k(i)\) is the index of the position of node \(i\) in solution OptPath_k */
    # \(m_{rs} := m_{rs} + \frac{L_k}{L_k} /* This is equation (3) */\)
    # \(m_{sr} := m_{sr} - \frac{L_k}{L_k} /* This is equation (4) */\)
    # EndFor
    # counter := counter + 1
    # If (counter mod \(v\) == \(u\))
    # For \(i := 1\) to \(w\)
    # \((r, s) = \arg \max_{(j, k) \in A, (j, k) \notin P} \{m_{jk}\}\)
    #   If \((m_{rs} \geq 0)\)
    #     \(R := R \cup (r, s)\)
    #   Else
    #     \(i := w /* Forcing the exit from the For loop */\)
    #   EndIf
    # EndFor
    # EndIf
   # EndFor

   Let \(L_{best}\) be the shortest \(L_k\) from beginning and OptPath_{best} the corresponding path
   For each arc \((r, s) \in OptPath_{best}\)
   \(\tau_{rs} := (1 - \rho) \cdot \tau_{rs} + \rho / L_{best} /* This is equation (1) */\)

   EndFor

   If (Time > MaxTime)
   then
     Print \(L_{best}\) and OptPath_{best}
   else
     repeat Step 2
   EndIf

Figure 1. The HAS-SOP_{APC} algorithm. The omission of the steps marked with \# leads to the conventional HAS-SOP algorithm.
<table>
<thead>
<tr>
<th>Problem</th>
<th>Average results</th>
<th>Improvement over HAS-SOP (%)</th>
<th>Best results</th>
<th>Improvement over HAS-SOP (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Results</td>
<td>p = 30</td>
<td>p = 15</td>
<td>p = 1</td>
</tr>
<tr>
<td>n</td>
<td>r</td>
<td>p</td>
<td>HAS-SOP</td>
<td>HAS-SOP</td>
</tr>
<tr>
<td>200</td>
<td>100</td>
<td>1</td>
<td>91.8</td>
<td>88.8</td>
</tr>
<tr>
<td>300</td>
<td>100</td>
<td>1</td>
<td>77.6</td>
<td>72.6</td>
</tr>
<tr>
<td>400</td>
<td>100</td>
<td>1</td>
<td>63.8</td>
<td>63.8</td>
</tr>
<tr>
<td>500</td>
<td>100</td>
<td>1</td>
<td>55.2</td>
<td>55.6</td>
</tr>
<tr>
<td>600</td>
<td>100</td>
<td>1</td>
<td>50.6</td>
<td>48.4</td>
</tr>
<tr>
<td>700</td>
<td>100</td>
<td>1</td>
<td>44.4</td>
<td>40.4</td>
</tr>
</tbody>
</table>

Averages for r = 100 and p = 1

| 3.72 | 1.86 | 6.63 | 1.80 |

Averages for r = 1000 and p = 1

| 1.01 | -0.34 | 1.73 | 0.25 |

Averages for p = 1

| 2.37 | 0.76 | 4.18 | 1.02 |

Averages for r = 1000 and p = 15

| 1.92 | 0.25 | 1.90 | 0.16 |

Averages for p = 15

| 3.21 | 0.57 | 4.12 | 2.49 |

Averages for r = 1000 and p = 30

| 2.57 | 0.41 | 3.91 | 1.32 |

Averages for p = 30

| 0.84 | 0.65 | 0.71 | 0.24 |

Averages for r = 1000 and p = 50

| 0.71 | 0.46 | 0.58 | 0.52 |

Averages for p = 50

| 0.78 | 0.66 | 0.65 | 0.38 |

Averages for r = 1000 and p = 60

| 0.05 | 0.05 | 0.07 | 0.08 |

Averages for p = 60

| 0.06 | 0.02 | 0.05 | 0.04 |

Overall averages

| 1.44 | 0.47 | 1.97 | 0.70 |