Predicting the performance of a parallel heuristic solution for the Steiner Tree Problem

Héctor Cancela cancela@fing.edu.uy
Ariel Sabiguero asabigue@ieee.org
Universidad de la República
Montevideo, Uruguay

Technical report INCO 03-01
ISSN: 0797-6410
February 2003

Abstract

Nowadays, there is an increasing number of computer intensive applications, which exceed the capacity of a standard stand-alone computer. An alternative is to parallelize the application and run it in a cluster; there has been much work in this sense, specially in platforms and tools to build a cluster from commodity components, and to develop parallel applications. One of the problems that subsist is the one faced by the analyst when designing a new application in this environment. He must solve the trade-off between the cost of building the cluster, and the application's running time; if he under-dimensions the cluster, the running time might be too long; if he over-dimensions it, the cost might not be acceptable.

This work presents an example of how analytical performance models can be applied in this context. In particular, we develop a parallel implementation of a combinatorial optimization heuristic for solving the Steiner Tree Problem, and a Petri net model which can be used to predict the running time of the application on a cluster of PCs, on the basis of measurements on stand-alone equipment. The model is validated experimentally, showing that it adequately predicts optimistic and pessimistic bounds for the measured running time.

Keywords: performance estimation, parallel applications, Petri net models, combinatorial optimization, Steiner Tree.

1 Introduction

Nowadays, there is an increasing number of computer intensive applications, which can not be computed in a reasonable time in a stand-alone computer. An alternative is to parallelize the application and run it in a cluster; there has been much work in this sense, specially in the development and testing of platforms and tools which permit to build a cluster from commodity components, and to efficiently develop a parallel application [2].

One of the problems that subsist is the one faced by the analyst when designing a solution for a new application in this environment. he must solve the trade-off between the cost of buying the components and building the cluster, and the running time of the application; if he under-dimensions the cluster, the running time might be too long; if he over-dimensions it, the cost might not be acceptable. Also, he has to design the architecture of the application, which has implications on the cluster performance.

This problem can be tackled by developing analytical performance models, and using them to evaluate the performance of different alternatives for the solution design [5]. A formalism that has

proved very useful for this task is Petri nets. The success of Petri nets is mainly due to the simplicity of the basic mechanism of the model, which on the other hand present drawbacks on the description of large systems. Basic Petri net models can be extended introducing the notion of time; in particular, we are interested in models where random variables are used to specify the time behavior of the model.

An important application area for parallelism is the solution of optimization problems which arise in designing different kind of systems. The objective of this work is to show how a particular combinatorial optimization problem can be solved by a parallel application, and how a Petri net model can be used to predict the performance of this parallel application on a cluster of heterogeneous computers, on the basis of measures in stand-alone equipment.

The combinatorial problem under study is the Steiner Tree Problem in Graphs, which is described in Section 2. Section 3 presents a heuristic method which can be used to solve this problem, together with a master-slave parallelization scheme. The Petri net model is developed in Section 4. Numerical experiments with the model are reported in Section 5, comparing the performance estimates obtained with the model and measured data of real parallel executions. Finally, conclusions and future work are discussed in Section 6.

2 The Steiner Tree Problem in Graphs

The Steiner Tree Problem in Graphs (STPG) is a well-known combinatorial optimization problem, which consists of finding a connected sub-network that covers a subset of nodes of a given network with minimum cost. STPG can be used to model many different practical applications. Computing and communication examples are VLSI design, FPGA routing, communication networks topology optimization, multicast routing, etc; but STPG has also applications in many other areas, as diverse as for example phylogeny studies.

A formal definition of the problem follows. Let $G = (V, E)$ be a connected undirected graph, where V is the set of nodes and E denotes the set of edges. Let w be a non-negative weight function that associates the set of edges with positive real values and let K be a subset of nodes called terminal nodes. Let k be the number of terminal nodes. The Steiner problem STPG(V, E, w, K) consists of finding a minimum weight connected subgraph of G spanning all terminal nodes in K. The solution of STPG(V, E, w, K) is a Steiner minimal tree (SMT). The non-terminal nodes that are part of the solution are called Steiner nodes. This problem is inherently complex (from the point of view of computation). Karp [6] proved that STPG is NP-Complete in the general case. There is much continuing work on this problem and its variants [3]. One approach is to develop heuristics which can give good solutions in reasonable times; further improvements in execution times can be obtained if these heuristics are parallelized [9].

3 The SN metaheuristic

SN is a new combinatorial optimization metaheuristic, recently introduced in [13]. The SN metaheuristic decomposes a combinatorial optimization problem into a set of decision subproblems, which are solved heuristically; with this information, the original problem is transformed into a new, less complex one. This process is iterated until we reach a case that is trivial or can be exactly solved; resulting in a (near optimal) solution for the original problem. Each of these iterations is highly decoupled and can be easily parallelized, as suggested on [13]. The idea behind the SN metaheuristic is both simple and powerful as it turns optimization problems into decision ones extracting information for successive decisions from possibly inaccurate results of heuristic resolution. The following pseudo-code describes the basic scheme of SN:

```
Input: problem Q
While it is possible to divide Q into subproblems q_1, ..., q_n do
    For i from 1 to n do
        Solve q_i heuristically to obtain a solution s_i and a confidence measure cv_i
    End For
    Obtain i with maximum cv_i
    Modify Q using the information provided by q_i and s_i
End While
```

The method assumes that in each step, the problem Q can be divided into n decision subproblems $q_1, ..., q_n$. Using the heuristic we can obtain a certain solution s_i with an associated confidence value cv_i. After determining the optimal heuristic solution, the problem Q is modified into Q' that should be simpler, according to some problem size or complexity metric. As the supbproblems $q_1, \ldots, q_n$ are unrelated, in [13] it is mentioned that it should be possible to easily parallelize the algorithm, by assigning the heuristic solution of each to a different CPU.

We propose the following pseudo-code for solving the STPG, inspired by the SN heuristic. In this case, the subproblems consist in evaluating smaller STPG problems to help deciding whether a given non-terminal node belongs or not to the optimal solution.

```
Input: graph G = (V, E), weight function w, terminal set K

While |K| < |V| do
    For i in V\K do
        sol_YES = STH(G, K ∪ {i})
        sol_NO = STH(G − {i}, K)
        if sol_YES < sol_NO
            s_i = Yes; cv_i = sol_YES
        Else
            s_i =No; cv_i = sol_NO
    End For
    Obtain i with minimum cv_i
    If s_i =Yes then
        K = K ∪ {i}
    Else
        V = V\{i}
End While
Solution = G
```

At each step of the iteration we have $2m$ Steiner Tree subproblems, which are solved by a heuristic STH which gives fast results, with $m = |V| - |K|$ non-terminal nodes; as either V decreases or K increases at each iteration, m monotonely decreases, and the algorithm ends in a finite number of steps. The method performance (both in solution quality and computational requirements) depends on the STH heuristic, which is used to solve the intermediate subproblems; this is usually a simple, non-iterative method, as its execution time should be as small as possible. In our initial tests we used a very simple heuristic that we called DijkstraPlusPrune. It consists of picking randomly an initial node and determining the Dijkstra tree from that node. It is a solution because it is a tree that covers all the terminal nodes, since the Dijkstra tree covers all nodes in a connected graph. After finding the tree, we proceed pruning all non terminal nodes with degree one, that means, unnecessary nodes for the connectiveness of the terminal nodes. The cost of determining the Dijkstra tree is $O(n^2)$.

Since every decision takes $2m = 2(n-k)$ executions of the heuristic, we can determine that the order of each decision is $O(n^3)$. Now we can see that the order of execution of the whole SN metaheuristic is $O(n^4)$. Being more precise, the execution order is $O((n-k)^2 n^2)$ that is equivalent to $O(n^4)$ when $n >> k$. This is the most general case in STP resolution. We can also see that if n and k are of the same order of magnitude, the execution time will have order $O(n^2)$ approximately.

The way we parallelized the algorithm, the same suggested by SN authors, consists in running in parallel all the heuristics that cooperate to take each decision. The most simple approach is to take $2m$ single CPU heuristics and run them in parallel. This is the approach we followed. We used single-threaded heuristics to solve each decision problem, while running sets of them in parallel. The metaheuristic imposes a limit in the speedup: we can not spawn more than $2m$, twice the number of non-terminal nodes, problems at the same time, even though there are $m(m-1)$ heuristics to solve. We have to take one decision at a time so as to build the solution. Each iteration will require less computations to solve, until the last decision when we have to decide if the remaining node shall be present or not in the solution, leading us to computing two heuristics. We can see that the usage of computational resources decreases in time.

This decomposition corresponds to a master-slave or task-farming paradigm. We can identify two entities: master and slaves. The master is responsible for decomposing the problem into small tasks, distributing them among the slaves, collecting results and assemble the problem solution. Slaves perform a simple sequence of steps: get a message with the task, process the task and send the result to the master. In this case, there is no communication among slaves. This kind of problems are easily scalable (adding more slave CPUs) and their speedup is quasi-linear. The work is statically decomposed and dynamically distributed.

4 Performance model

The mathematical model selected for this study is Petri nets, which were introduced by C. A. Petri en 1962. The theoretical grounds for Petri net theory have been deeply investigated and today build a formal structure with a well assessed theory and a broad range of applications (in particular in the field of performance evaluation [1, 8, 10]. The success of Petri nets is mainly due to the simplicity of the basic mechanism of the model, which on the other hand present drawbacks on the description of large systems. Basic Petri net models can be extended introducing the notion of time; in particular, we are interested in models where random variables are used to specify the time behavior of the model. After researching multiple available tools for working with Petri nets we selected the UltraSAN tool [12, 11], from the Center for Reliable and High Performance Computing, University of Illinois (available at http://www.crhc.uiuc.edu/UltraSAN as of 2002-09-27). This tool proved very useful in our research. UltraSAN provides a graphical editor for the networks and a set of solvers that allows general problem resolution. UltraSAN is based on Stochastic Activity Networks, a variant of Petri net model which shares the basic elements with the standard Petri nets: places, arcs and transitions; adding the concepts of input and output gates, which allow to specify a more detailed logic governing the behavior of the transitions, and mechanisms for modelling rewards associated both to states and to transitions of the network.

Figure 1 shows the Petri net model (with UltraSAN notation) for the parallel method for solving the STPG, when running on a cluster with two different types of cpus (the cpus are classified in homogeneous groups according to their characteristics: memory, processor type and speed, bus). The model includes the following places:

- Problem_Input: has initial marking $m = |V| - |K|$, corresponding to the size of the problem to be solved. As the Petri net state changes in time, this place will hold the size of the subproblems currently considered.

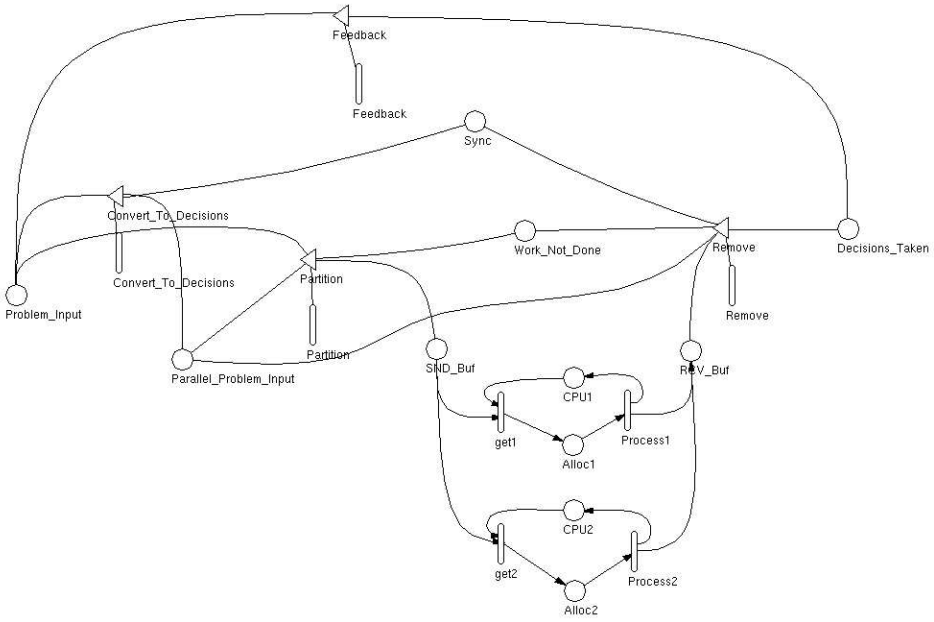

Figure 1: Stochastic Activity Network model for SN application

- Parallel_Problem_Input: this place has a token prior to the subdivision of the problem in decision subproblems to be solved in parallel.
- Sync: this place will be used to detect when all subproblems have been solved at an iteration of the heuristic; it will hold the number of subproblems which were defined at each iteration; when this number equals the number of subproblems solved (RCV_Buf), the iteration is completed.
- Work_not_done: the tokens in this place represent the subproblems which will be solved at a given iteration.
- SND_Buf: this place holds a token for each subproblem waiting to be solved.
- RCV_Buf: this place holds a token for each subproblem already solved.
- CPU1: this place corresponds to idle cpus of the first class. Initially its marking corresponds to the number of cpus of this class in the cluster.
- Alloc1: this place will hold a token for each cpu of the first class which is busy processing a subproblem.
- CPU2, Alloc2: these places have the same semantics as CPU1 and Alloc1, corresponding to the second group of cpus. Eventually there could be places CPU3, Alloc3, etc, for other classes of cpus.
- Decisions_Taken: this place will hold a token for each "decision taken", i.e. for each iteration of the main loop succesfully completed.

We have also the following transitions:

- Convert_To_Decisions: this transition consumes a token from Problem_Input, and creates one token in place Parallel_Problem_Input and MARK(Problem_Input) tokens in place Sync. This corresponds to the outer loop of the SN heuristic.

- Partition: converts the token in Parallel_Problem_Input into $2m$ = 2MARK(Problem_Input) tokens in SND_Buf, corresponding to the number of subproblems to be solved in each iteration (as the tokens in Problem_Input diminish at each iteration, so do the number of subproblems that are considered here).

- Get1: this transition takes a token from SND_Buf and from CPU1, and generates a token in Alloc1; corresponding to assigning an idle cpu of the first class to a subproblem waiting to be solved.

- Process1: this transition consumes a token from Alloc1, and generates a token in CPU1 and in RCV_Buf; corresponding to a cpu finishing the evaluation of a subproblem, and being freed up.

- Get2, Process2: these transitions have the same semantics as Get1 and Process1, corresponding to other cpu classes. Eventually there could be transitions Get3, Process3, etc, for other classes of cpus.

- Remove: when the iteration is completed and all subproblems have been evaluated (RCV_Buf=Sync) this transition empties Sync, Work_Not_Done and RCV_Buf, and puts a token into Decisions_Taken.

- Feedback: when the problem has been completely solved (Decisions_Taken= m), this transition takes m tokens from Decisons_Taken, and puts m tokens into Problem_Input. This in fact restarts the whole process, as if the problem once finished was run again from the beginning. The idea is to transform a terminating system into a recurrent one, in order to evaluate steady state measures (which are more easy to obtain numerically) instead of transient ones.

We can see that the resolution of the whole problem consists in decisions, one for each non terminal node. There exists an implicit synchronization after each of these decisions while the master determines the best one.

We are interested in computing the total execution time, since the optimization method starts (marking with m tokens in place Problem_Input) until all iterations have been finished (marking with m tokens in place Decisions_Taken). We observe that the Petri net has a cyclic behavior, going from the first of the aforementioned markings to the last one, and afterwards enabling transition Feedback which transforms the last marking into the first one. Let us call T_c to the total cycle time, T_p to the problem time and T_f to the Feedback transition time (which is a fixed parameter of the net). The total cycle time is the sum of both problem time and the feedback time: $T_c = T_p + T_f$. We define an UltraSAN performance variable, called Probability, as an impulse reward, which adds 1 each time that the transition Feedback is activated. UltraSAN can compute the steady state value of Probability averaged over elapsed system time; as Probability= $1/T_c$, from this value we can recover T_c. As T_f is a known parameter, we can from these two values compute T_p, which is the total execution time for the optimization method.

One important remaining point is to specify the probability distribution function of the timed transitions of the model. This has an important effect on the system behavior, as distributions with the same mean value for the transition time but different variance will result in different total execution times for the system.

For the effects of this study, we propose employing two different distributions, the exponential and the deterministic (constant) one. Both distributions are completely determined by one parameter, the mean value for the transition time. The deterministic distribution corresponds to an optimistic case, as a deterministic transition time can be obtained only when the system is not subject to any random variation or external influence. It is also a well known fact of queuing theory that deterministic systems are the ones with best behavior, in terms of waiting times and system occupation. The exponential distribution, on the other hand, has a quite large variance (standard deviation equal to the mean of the distribution). This is then a pessimistic estimation, corresponding to systems with very high

variability in execution times of individual steps, which in turn will result in lower overall system performance.

5 Numerical experiments

5.1 Implementation details

In order to validate the model developed in the previous section, we implemented the parallel optimization method for the STPG and compared the predicted versus actual execution times. The optimization method was implemented using Java, a younger alternative to Object Oriented Programming [4]. Java is inherently platform independent and the bytecode can be run in any system where exists a Java Virtual Machine. A significant drawback is that the JVM bytecode, is interpreted, and thus, slower than the object code produced out of a C++ compilation. Java supports a Remote Objects paradigm, which was the tool used for programming and running our heuristics on remote virtual machines over different systems.

The optimization algorithm is implemented as a method in a class. It receives a weighted graph, the subset of terminal nodes, a heuristic, a criteria and returns the resulting tree. The metaheuristic converts the problem into a succession of sets decision problems: at each iteration, the "best" decision is taken, until the solution is found. This general metaheuristic can be applied to almost any problem that accepts a compositional solution. In the case of the STPG, the decision consists of determining at each stage for every non terminal node will or will not be part of the solution. Each individual decision is taken considering the heuristic applied to the graph considering that the decision has already been taken. The "best" option, according to the heuristic, is taken at each stage, fixing one non-terminal node as a Steiner node or removing it from the solution. Java makes the use of multithreading simple. We use a thread to control each remote object. The threads access a common set of graphs to solve, pick some of them, submit the job to the remote object, gather the result and send more jobs until no other graphs are there for solving using the heuristic. The resolution of the heuristic takes place remotely, but the synchronization and access to the information is solved within the same Virtual Machine, which makes it simpler to coordinate execution. Instead of having different programs running in different memory spaces, we have a set of threads running with the same permissions in the same virtual machine. The set of remote objects do not interact amongst them, but through the master process.

5.2 Test cases and platform

We based our tests on problems from the B series of the SteinLib [7], a very well known library of standard instances for the Steiner Tree Problem. Table 1 resumes some relevant characteristics of the problems. The first three columns show the number of nodes, the number of edges and the number of terminal nodes for each of the problems. These parameters determine the complexity of the network. The other column, Subproblems, shows the number of heuristics that have to be solved in the worst case.

We made two series of tests. The first one was run on a very small cluster, consisting of five machines configured as follows: one Pentium III 933Mhz, 512MB RAM, Linux 2.4.18; three Pentium II 400MHz, 256 MB RAM; and one Pentium II 400MHz, 256 MB RAM Windows NT 4.0. The virtual machine used is Sun Microsystem's Java 2 Standard Edition 1.4.0 (build 1.4.0-b92) on Windows and Linux systems.

The second series of tests were run on a computer lab at the Engineering School of our institution . This room is equipped with two different generation of computers, all of them running Solaris OS. There are ten PCs with Pentium IV of 1,6 GHz and 512 MB of RAM, and seventeen additional ones

Name	$\|V\|$	$\|E\|$	$\|K\|$	Subproblems
B01	50	63	9	1640
B02	50	63	13	1332
B03	50	63	25	600
B04	50	100	9	1640
B05	50	100	13	1332
B06	50	100	25	600
B12	75	150	38	1332
B18	100	200	50	2450

Table 1: Test problems from Steinlib

with Pentium II processors of 350MHZ and 64 MB of RAM. The virtual machine used is Java 2 Standard Edition 1.4.0_01.

5.3 Parameter fitting

To estimate the parameters of the model, we run the master and the slave processes on one machine of each generation for each problem in the series. The original codes were slightly modified so as to get timing information. The slave processes were coded so they measured the time elapsed for each invocation. The execution conditions were kept as stable as possible. In the case of the computer lab, the tests were run during a couple of weekends, to minimize student load on the machines (even though students were not forbidden to use the computer lab, and actually a few did so while the tests were being executed).

The distribution functions that need to be defined according to empirical data are the ones associated with the following timed activities: Convert_To_Decisions, Partition, Remove, Get and Process.

As network throughput was not a problem and the computation/communication ratio was very high, Get activities had negligible times (less than 1 ms.).

The activities Convert_To_Decisions and Partition took a significant time in the five-machine experiments. For the experiments in the computer lab, we modified the partitioning code in the master process to avoid storing in memory all problems before starting the resolution. This fact proved important when addressing big problems as with the first version the virtual machine run out of memory. In this scenario, the serial processing at the master is restricted to the first decision creation; afterwards all the processing is done in parallel.

5.4 Predicted versus measured execution times

Table 2 shows the optimistic and pessimistic estimations obtained with the proposed model for the first six problem instances from the B series of the SteinLib (named b01, b02, ..., b06), as well as measured execution times for these same cases, for the five-machine cluster. Columns 1 and 2 correspond to the pessimistic and optimistic estimations for mean total execution time. Columns 3,4 and 5 show the minimum, average and maximum values for actual execution time (the problem was run three different times on the same platform, to have some feeling of the variability of the results).

We can see that in all cases, the average value is well within optimistic and pessimistic estimators. What is more, the interval defined by the observed worst execution time and the best one is within the interval defined by those estimators. The optimistic and pessimistic estimators have the same order of magnitude, resulting in a prediction interval relatively tight.

The results corresponding to the larger cluster (twenty-seven machine computer lab experiments) are shown in Table 3. We show the estimations obtained for the same six problem instances from the B series of the SteinLib (b01 to b06), as well as two additional ones (b12, b18), corresponding

Problem	Optimistic prediction	Min. observed	Average observed	Max. observed	Pessimistic prediction
b01	122	218	232	259	345
b02	104	191	198	207	288
b03	53	114	120	127	170
b04	165	309	329	342	469
b05	140	257	281	306	402
b06	75	184	187	193	229

Table 2: Predicted and measured execution times (seconds) in five-machine cluster

Problem	Optimistic prediction	Min. observed	Average observed	Max. observed	Pessimistic prediction
B01	20	94	99	105	658
B02	17	87	90	93	616
B03	9	57	60	63	317
B04	27	126	133	138	886
B05	25	109	126	155	803
B06	12	85	95	107	519
B12	94	410	431	446	2933
B18	450	1566	1682	1895	12079

Table 3: Predicted and measured execution times (seconds) in larger cluster

to bigger instances. Again columns 1 and 2 correspond to the pessimistic and optimistic estimations for mean total execution time, and columns 3,4 and 5 show the minimum, average and maximum values for actual execution time. As can be seen from the table, the measured values are within the optimistic and pessimistic estimations obtained with the analytical model.

6 Conclusions

The main objective of this work was to apply Petri net models to performance evaluation of parallel combinatorial optimization applications running on clusters of PCs.

In particular, a Petri net model was developed which represents the behavior of a parallel method inspired on the SN metaheuristic for solving the Steiner Tree Problem. This model was used to obtain both pessimistic and optimistic performance estimators, by modelling individual resolution times with exponential and deterministic probability distribution functions respectively. The performance estimator was the expected total execution time for solving a given problem instance. This estimator was compared with the actual execution time measured for the parallel application in two different heterogeneous clusters, with good results; this validates both the model, and the general assumption of optimistic behavior associated to deterministic distribution functions and the pessimistic behavior associated with exponential distribution functions.

We found several practical problems in the process of solving built models regarding to the Petri network resolution. One issue was the model state space, which grows very fast with problem size (Table 4 shows the number of states generated for each problem instance). Related to this fact is the growth in resolution time.

This shows also the importance of using high level models, as more detailed ones would be difficult to evaluate, due to high model evaluation times. We believe that the model proposed is detailed enough to capture the main characteristics of the parallel application, without loosing the model evaluation feasibility.

Problem	State space size
B01	189351
B02	147565
B03	54085
B04	189351
B05	147565
B06	54085
B12	136176
B18	262418

Table 4: Analytical model state space size

It is important to note that the models presented themselves do not optimize the configuration of a parallel machine for a certain purpose but can help the designer deciding this configuration. In this environment the tools could be useful helping designers either determining that existing hardware is enough for performance requirements or for justifying investment on newer hardware. This kind of models can also help to explore the convenience of different algorithms for solving a given problem on certain hardware; as they can provide performance estimators for different algorithms solving the same problem on the same hardware.

Future work could explore in more detail the influence of the different probability distribution functions in the model. On one hand, it could be possible to empirically determine what distribution functions best adjust to measured execution time for each of the steps of the method, and by fitting these values into the model a more accurate estimate for total execution time could be obtained. An additional refinement of the model could take into account that the resolution time for the subproblems decreases in mean at each iteration of the optimization method. This happens because the network size decreases at each iteration, and the running time complexity of the heuristic used to solve the subproblems is directly related to the number of nodes and edges of the network. Even though we found that the average on these measured times is suitable for our studies, it is possible to use other functions to estimate problem resolution time expected values every time. It might be of interest to use a function of the number of decisions or invocations instead of a constant one. Such a function would fit better the gathered data and should be a better model of the reality.

References

[1] M. Ajmone Marsan, G. Balbo, and G. Conte. *Performance Models of Multiprocessor Systems.* The MIT Press, 1988.

[2] Rajkumar Buyya. *High Performance Cluster Computing.* Prentice Hall, 1999.

[3] D-Z. Du, J.M. Smith, and J.H. Rubinstein. *Advances in Steiner Trees.* Kluwer, 2000.

[4] B. Eckel. Thinking in java, Accessed at http://www.eckelobjects.com (20/9/2002).

[5] B. R. Haverkort, R. Marie, G. Rubino, and K. S. Trivedi, editors. *Performability Modelling: Techniques and Tools.* John Wiley & Sons, 2001.

[6] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Numerical Solution of Markov Chains.* Plenum Press, New York, 1972.

[7] T. Koch, A. Martin, and S. Voss. SteinLib: An updated library on Steiner Tree Problems in graphs. Technical report 00-37, Konrad-Zuse-Zentrum fur Informationstechnik, Berlin, 2000.

[8] Christoph Lindemann. *Performance Modelling with Deterministic and Stochastic Petri Nets.* John Wiley & Sons, 1998.

[9] S. L. Martins, C. Ribeiro, and M. C. Souza. A parallel GRASP for the Steiner problem in graphs. In *Workshop on Parallel Algorithms for Irregular Structured Problems*, pages 285–297, 1998.

[10] J. F. Meyer and W. H. Sanders. Specification and construction of performability models. In B. R. Haverkort, R. Marie, G. Rubino, and K. S. Trivedi, editors, *Performability Modelling: Techniques and Tools*, chapter 9, pages 179–222. John Wiley & Sons, 1996.

[11] W. H. Sanders, W. D. Obal II, M. A. Qureshi, and F. K. Widjanarko. The UltraSAN modeling environment. *Performance Evaluation*, 24(1):89–115, October-November 1995.

[12] W. H. Sanders and W.D. Obal, II. Dependability evaluation using UltraSAN. In *FTCS-23 Digest of Papers*, pages 674–679, Toulouse, France, June 1993. IEEE Computer Society Press.

[13] S. Urrutia and I. Loiseau. A new metaheuristic and its application to the Steiner Problems in graphs. In *XXI International Conference of the Chilean Computer Science Society (SCCC'01)*, Punta Arenas, Chile, November 2001.