Two Approaches to Simulated Annealing for Uncapacitated Facility Location Problems

[a]M. Emin Aydin[1], [b]Vecihi Yigit, [a]Terence C. Fogarty

[a] *South Bank University, School of Computing, Information Systems and Maths, 103 Borough Road, London, SE1 0AA, UK*
[b]*Gazi University, Dept. of Industrial Engineering, Ankara, Turkey*

Abstract

The aim of this paper is to examine two new simulated annealing approaches for Uncapacitated Facility Location (UFL) problems with some useful comparisons with the latest genetic algorithm approach (Jaramillo et al, 2002) to the problem. As it is mentioned in various publications, the simulated annealing approach may be very successful in terms of quality of the solution but not so impressive with respect to the CPU times. The approaches presented in this paper are seeking for utilising that power of the method in getting quality of solutions within shorter time. For this purpose, simulated annealing algorithms incorporated in evolutionary and parallelisation approaches in order to cut down the time needed.

Key words; facility planning and design, simulated annealing, multi-agent systems

[1] Corresponding author
Address: South Bank University, School of Computing, Information Systems and Maths, 103 Borough Road, London, SE1 0AA, UK
Tel: +44 (20) 7815 7414
Fax: +44 (20) 7815 7499
e-mail: aydinme@sbu.ac.uk

1. Introduction

Facility Location Problems that have NP-Hard nature have been playing an important role in operation research and manufacturing context. The un-capacitated facility location (UFL) problem, also known as simple plant location problem, is basically a member of the family of location problems. In a general form, the problem is to determine the optimal number of facility echelons, the number and location of facilities in each facility echelon, the assignment of distribution activities / commodities to facilities, and the allocation of customer demand among the facilities. The UFL problems have several applications, such as the bank account location problem, the clustering problem, economic lot sizing, vehicle routing, network design, distributed data and communication networks.

The mathematical formulation of these problems as mixed integer programming models has proven very fruitful in the derivation of solution methods. To formalise the UFL problems, consider a set $J = \{1,...,m\}$ of candidate sites for facility location, and a set $I = \{1,...,n\}$ of customers. Each facility $j \in J$ has a fixed cost f_j. Every customer $i \in I$ has a demand b_i, and c_{ij} is the unit transportation cost from facility j to customer i. Without a loss of generality we can normalise the customer demands to $b_i = 1$. The problem is formulated in the following way:

$$Z = \min \sum_{i=1}^{n} \sum_{j=1}^{m} c_{ij} x_{ij} + \sum_{j=1}^{m} f_j y_j \qquad (1)$$

subject to:

$$\sum_{j=1}^{m} x_{ij} = 1, \qquad \text{for } \forall\ i \in I; \qquad (2)$$

$$0 \le x_{ij} \le y_j \quad \text{or} \quad y_j \in \{0; 1\}; \quad \text{for } \forall\ i \in I \text{ and } \forall\ j \in J; \qquad (3)$$

where x_{ij} represents the quantity supplied from facility j to customer i, y_j indicates whether facility j is established ($y_j = 1$) or not ($y_j = 0$). The constraint (2) makes sure that all demands have been met by the open sites, and the constraint (3) is to keep integrity. Since it is assumed that there is no capacity limit for any facility, the demand size of each customer is ignored, and therefore constraint (2) established without considering demand variable ($b_i = 1$).

By this model, there has to be decided for (i) the number of facility sites to be established and (ii)- the quantities to be supplied from facility j to customer i such that the total cost (including fixed and variable costs) is minimised.

UFL problems have been studied for many years and thus there is a very rich literature in operations research (OR) for these kinds of problem. Since they have a NP-Hard nature, the larger the size of the problem, the harder to find the optimal solution and furthermore, the longer to reach reasonable results. During the past three decades, these UFL problems have been triggered and examined extensively (Kratica et al., 2001) by various attempts and approaches. The presentation of all-important contributions relevant to UFL problems can be summarised as follows. There are mainly two categories of approaches to this type of problems: classical OR, (branch and bound, primal and dual ascent methods, linear programming and Lagrangean relaxation algorithms) and meta-heuristic based methods. The Dualoc algorithm (Erlenkotter, 1978) is one of the most respected methods based on OR approaches as the fastest one for UFL problems for a long time. It is based on a linear programming dual formation (LP dual) in condensed form that evolved in simple ascent and adjustment procedure. If ascent and adjustment procedures do not find the optimal solution, Branch-and-Bound (BnB) procedure completes the solution process.

Guignard (1985) proposed to strengthen the separable Lagrangean relaxation of the UFL problems by using Benders inequalities generated during a Lagrangean dual ascent procedure. The coupling of that technique with a good primal heuristic could reduce the integrity gap. Simao and Thizy (1989) presented a streamlined dual simplex algorithm designed on the basis of a covering formulation of the UFL problem. Their computational experience with standard data sets indicates the superiority of dual approaches. Koerkel (1989) showed how to modify a primal-dual version of Erlenkotter's (1978) exact algorithm to get an improved procedure. The computational experience with large-scale problem instances indicated that speedup to Dualoc is significant (more than one order of magnitude). Coon and Cornuejols (1990) present a method based upon the exact solution of the condensed dual of LP relaxation via orthogonal projections. In Holmberg (1995) and Holmberg and Jornsten (1996) a primal-dual solution approach based on decomposition principles is used. They fixed some variables in the primal sub-problem and relaxed some constraints in the dual sub-problem. By fixing their Lagrange multipliers, both of these problems become easier to solve than the original one. The computational tests proved the advantageous in comparison to the dual ascent method of Erlenkotter.

On the other hand, Alves and Almeida (1992) have compared the results of four versions of their simulated annealing algorithm with those of some well-known heuristic methods. The quality of the solutions is very good, but the computing time is significantly longer in comparison to the others.

TANN (Tabu Neural Network) by Vaithyanathan (1996) integrated an analogue version the short term memory component of taboo search with neural networks to generate a massively parallel, analogue global search strategy, which was applied on UFLP instances of small dimensions (10x10, 20x20 and 30x15). The obtained solutions have errors more than 20% from optimal ones.

Kratica et al. (2001) have applied genetic algorithms to UFL problems to solve 1000x1000-sized customer-facility. They considered many benchmarks within the literature to be solved by their algorithm in addition to their own such large sized problems. They compared their results with Erlenkotter's dual-based algorithm (DUALOC) showing that their algorithm is much more efficient than DUALOC for the problems larger than 100x1000. Although DUALOC has better results for some benchmarks, it is worse in time consumption. Jaramillo et al. (2002) applied a genetic algorithm that is mainly based on the operators that Beasley and Chu (1996) applied in their genetic algorithms for covering problems. They compared their results with a Lagrangean relaxation algorithm presented by Beasley (1993).

The aim of this paper is to examine two new simulated annealing approaches for UFL problems with some useful comparisons with the latest genetic algorithm approach (Jaramillo et al., 2002) to the problem. Although Kratica et al. (2001) have proposed a very similar approach regarding exploitation of genetic algorithms and the way they are implemented, Jaramillo et al. (2002) give a fresher and less time consuming approach with which it is much fairer to make a comparison. On the other hand, the approach Kratica et al. (2001) proposed looks more time consuming then Jaramillo et al. (2002) in terms of the number of operations considered in the implementation. As is mentioned previously in considering the work of Alves and Almeida (1992), simulated annealing approaches may be very successful in terms of quality of the solutions but not so impressive with respect to the CPU times. The approaches presented in this paper are seeking for utilising of the power of the method in getting quality of solutions within shorter times. For this purpose, simulated annealing algorithms incorporated by evolutionary and parallelisation approaches to cut the time needed.

The rest of this paper is organised as follows. The benefits of the evolutionary approach to simulated annealing are discussed in Section 2. The way in which simulated annealing parallelised is presented in Section 3. The experimental results and related discussions are given in Section 4 and finally the conclusion in Section 5.

2. Simulated Annealing

Simulated annealing can be viewed as a probabilistic decision making process in which a control parameter called temperature is employed to evaluate the probability of accepting an uphill move (in the case of minimisation). Suppose that s_n, s_n' and s_{n+1} denote the solution tackled (moved from) in the nth iteration, the solution moved to in the nth iteration, and the qualified solution for the $n+1^{th}$ iteration, respectively. s_n' is yielded by moving from s_n by the way of the neighbourhood function. The new qualified solution is determined as follows:-

$$s_{n+1} \longleftarrow \begin{cases} s_n' & \Delta s < 0 \\ s_n' & e^{-\Delta s / t_n} > \rho \\ s_n & otherwise \end{cases}$$

where $\Delta s = s_n' - s_n$, ρ is the random number generated for making a stochastic decision for the new solution and t_n is the level of temperature (at n^{th} iteration) that is cooled through this optimisation process by a particular cooling function, $T = f(t_n)$. That means, in order to qualify the new solution (s_n') for the next iteration, it should be either better then the old one (s_n) or worse but satisfied the stochastic rule to promote it. The satisfaction of the stochastic rule is the main idea behind simulated annealing in which the probabilistic decision is made to prevent the optimisation process from sticking in possible local minima. The probability of making such a decision under the circumstances of a Δs at temperature t_n is denoted by $e^{-\Delta s/tn}$. Clearly, as the temperature is decreased by T, the probability of accepting a large decrease in solution quality decays exponentially toward zero according to the Boltzman distribution. Therefore, the final solution is near optimal when the temperature approaches zero.

2.1 Representation and neighbourhood function

UFL problems are usually represented in a binary way in which the open facilities are denoted by 1 and closed ones by 0. In order to move from one solution to another, the digits are briefly converted to each other. Depending on the heuristics employed, this conversion is done either by a neighbourhood function, as happens here, or a genetic operation. We represent the state of solutions by a list of integers, say F, denoting the enumerated open facility, (f_i). For instance, $F=\{0,1,4,12\}$ means the open facilities are Number 0, 1, 4 and 12, the rest are considered as closed. As we denote the particular solution state at time t with s_t, one state of solution will be:-

$$s_t = \{F\} = \{ f_i / \quad 0 < i \le m \quad \} \quad \text{where } m \text{ is the maximum number of facilities}$$

The neighbourhood function that we employed allow us to modify the states by three different operations: (1) change one integer on the list with another possible one; (2) add up a new integer to the list; (3) remove one integer from the list, meaning exchange the state of one facility with another, or open a new facility or close one open facility, respectively. It allows applying only one of these operations at once. We select one operation randomly according to the following rule:-

$$operator \longleftarrow \begin{cases} Exchange() & (\lambda(F)=1 \cap 0 \le \rho < 0.7) \cup (\lambda(F)>1 \cap 0 \le \rho \le 0.5) \\ Add() & (\lambda(F)=1 \cap 0.7 \le \rho < 1) \cup (\lambda(F)>1 \cap 0.5 \le \rho \le 0.7) \\ Remove() & (\lambda(F)=m) \cup (\lambda(F)<m \cap 0.7 \le \rho \le 1) \end{cases}$$

where $\lambda(F)$ is the length of the list, ρ is a uniformly generated random number, and m is the maximum number of facilities as usual. By applying this function, we move to a neighbouring state. This is a preventive neighbourhood function that keep the solution feasible by letting the operators manipulate as they convenient. The convenience of one situation is determined by both the length of the list and the random number ($\lambda(F)$, ρ). For instance, if $\lambda(F)>1$ then any of the operations can be selected according to ρ, on the other hand, if $\lambda(F)=m$ then only *Remove()* operator is allowed to operate.

2.2 Modular Simulated Annealing (MSA)

The modular simulated annealing (MSA) algorithm partitions the SA algorithm into shorter slices to be implemented in various configurations together with different methods and environments. The idea behind modular SA is to have a more uniform distribution of random moves along the SA procedure. In fact, SA provides the solution process by exponentially distributed random moves such that each random move starts a new hill climbing process to reach the global minimum. However, the exponential nature of this distribution may not help to rescue the solution from local minima as in the case, when SA is applied to very difficult combinatorial optimisation that need more random moves, especially in the latter part of the optimisation process. But the probability of having a random move at that stage is so low as to make it longer to reach the global optimum.

Suppose that we have a problem that has a landscape like ℓ and E_0 as one of very strict local minima. Beside this, suppose we run a simulated annealing algorithm that sticks in E_0 under some initial conditions. Most of time, getting stuck in such local minima happens in the later stages of runs, when the cooling values are quite low, and therefore the probability moving to a rescuable neighbour approaches 0. In order to prevent a run of the algorithm, it is required to

relax the strict conditions to let the algorithm proceed by jumping to another branch of the solution tree that does not lead to E_0. A multi-start run of an algorithm is more helpful to relax these conditions rather than a single run. However, in that case, the time consumption will be another problem that may mitigate against the use of simulated annealing, if we do not shorten a single particular run. Thus, a reasonably shortened simulated annealing algorithm could be easily restarted, in order to allow to change the direction of solution path to a much more useful destination.

Another aspect of modular simulated annealing is to tackle a population of solutions rather than an individual. This is the issue of the effect of initial solutions on the optimisation process. Many works on seeking combinatorial optimisation problems with heuristics focused on the effects of initial solutions. When an initial solution has been chosen, there arises limited possible paths to proceed. Depending on the emerging new conditions, one can say whether a particular solution can lead to an optimal or useful near optimal solution with a certain probability. It is not wrong to say that a diverse population of solutions gives higher probability to reach an optimal or a useful near optimal solution than a single individual. Moreover, if useful selection and replacement strategies can be utilised, it will definitely help the process to improve the quality of solutions. So, for that reason, the modular simulated annealing algorithm is run on a population of solutions rather than an individual using various selection and replacement strategies.

Aydin and Fogarty (2001,2002) have applied a modular SA to some job shop scheduling benchmark problems that have variety of difficulty. The ideas of those works were to evolve a population of solutions by applying a modular SA constantly to the selected solutions.

A typical instance of the modular SA algorithm is presented in Figure 1. The algorithm is implemented to evolve a population of solutions running modular SA constantly up to a predefined number of iterations. First of all, a population of solutions is randomly initialised, and then, the number of iterations is set. After that, modular SA starts with the highest temperature (100), which is being cooled by a cooling coefficient (0.955) at each iteration. When the temperature cools to 0.01 short-term SA finishes with 200 iterations, which are counted to completion. The selected and optimised solution obtained through a modular SA is put back into the population. That is the end of one modular SA process. The succeeding cycle of evolution starts by selecting another solution randomly from the population. This process repeats until that total number of iterations is completed.

Begin

⇨ *Initialise* the **population**,

Repeat:

- *pick* one **feasible solution** (***old***),
- *se*t the **highest temperature** *(t=100),*

  repeat:-

  - *select* a particular **task**, *conduct* a move by **neighbourhood function**
  - *repair* the **new solution** (***new***)
  - ***if*** (***new-old***)<0 then *replace* ***old*** with ***new***
  - ***else***
    - *generate* a **random number** ***(r)***
    - ***if exp(-(new-old)/t)>r*** then *replace* ***old*** with ***new***
    - ***endif***
  - ***endif***
  - *t=t*0.955*

  until *t<0.01*

- *put* the **solution** back into the **population**

Until ***pre-defined number*** of **iterations**

End.

Figure 1: An instance of modular simulated annealing algorithm for evolution of a population

3 A Parallel implementation of MSA

As it is well known that there are two main ways to implement a system by parallel computation. One is by partitioning the whole data set into subparts and running the same algorithm on each of those subparts on multiple machines or processes. This could be called physical parallelism. The second one is more complicated in which the parallelisation is done on the algorithm itself rather than partitioning the data. That is called algorithmic parallelism. Since it is difficult to parallelise an ordinary SA in the sense of algorithmic parallelism, we have parallelised the system in the sense of physical parallelism.

3.1 Distributed Resource Machine (DRM)

DRM is the framework of a distributed problem solving environment that is one of two main part of the DREAM (Peacter et al.,2000) project (see Dream Web Site (2002) for more information) which deals with creating a multi island[2] (agent) based evolutionary computation environment running on a scalable network on Internet. The main idea is to have a peer-to-peer network of nodes spread on physically distributed computers. Each node has incomplete knowledge of the rest of the network and works as the container of all the agents running on that computer. Since DRM is an autonomous agent environment, the applications are implemented as multi agent systems. The environment has very good functionalities to develop

[2] Island means a particular agent that has some particular skills and a population of solutions in Evolutionary Computation context. Since DREAM project has an aim to utilise multi agent system in running evolutionary algorithms, we use this word instead of agent.

applications such that the agents would have good communication and limited mobility. (See Jelasity et al. (2002) for more information on DRM).

3.2 A parallel MSA based on multi-agent systems

As discussed in the previous section, MSA gives new opportunities to commence new valuable hill climbing processes in which the considered particular solution may give chances to change to a better situation. Therefore, the more time given to MSA to manipulate a particular solution, the easier it is to reach the global optimum. However, because of the reason we mentioned in the previous section operating on a single solution is not preferable for MSA. The idea is to bring the benefits of working on a population and in a multi-start fashion, together. But this is more time consuming. Suppose we have a population P sized as *ps.* In order to reach a preferable result, the SA needs to operate on a particular solution for a certain number of iterations, c, that last a certain CPU time, t. If one enhances the number of solution to operate on, that will linearly increase the number of iterations as well as the CPU time needed as:-

$$c_p = \sum_{i=0}^{np} c_i \quad and \quad t_p = \sum_{i=0}^{np} t_i$$

These two constraint MSA to work on a rather small-sized population. Unlike genetic algorithms, we need to work on better-designed small-sized populations to have the advantages of both the diversity of population and having more consideration by MSA run. However, it is difficult to have a good spectrum of solutions in small sized populations.

One of the possible solutions for this can be the consideration of parallel computing opportunities. The idea is to parallelise the application of MSA to distribute a rather bigger-sized population over more distributed agents to create more opportunities for letting MSA manipulate solutions for more times even within a shorter overall time. By means of parallelisation, it is more likely to have a shortening of both the total number of iterations (c_p) and the total consumed time (t_p). They will be shorter linearly in proportion to the number of agents, a. If we use the same notations, P could be partitioned into a as like P_0, P_1, …., P_a, denoting the sup-populations enumerated as $0, 1, \ldots, a$, respectively. Assigning each of these sub-populations to each particular agent, the total overall number of iterations will be still the same if we keep the number of iterations needed for a particular individual the same as before.

$$c_a = \sum_{j=0}^{a} \sum_{i=0}^{np} c_{ji} \quad and \quad t_a = \sum_{j=0}^{a} t_j$$

$c_a = c_p$ but $t_p \leq t_a$ because each CPU that the agents assigned may not have the same conditions and the communications among the agents will cost some more time. But it is clear to say that due to each agent running in parallel to the others, the concurrency cuts the time making the overall time as the maximum of spent times by all the particular agents. So $t_{overall} = max\{t_0, t_1, \ldots, t_a\}$.

The MSA algorithm has been implemented to run on Distributed Resource Machine (DRM). Since MSA has a modular nature, we easily designed this implementation for running on DRM as a multiple island[3] framework. It is required to partition the problem into subparts to be applied as a multi-island model for this purpose. So, we designed our islands with identical MSA algorithm and a small population of solutions where MSA operates on that population to evolve it towards an optimum value. The population uses a simulated annealing based replacement rule to promote new solutions over the old. The solution tackled per iteration is selected uniformly randomly, operated by MSA algorithm once and then is assessed to be replaced with its parent. Although the system allows that one randomly selected solution can attempt to migrate to another randomly determined island by a predefined period, we have not applied this operation in this particular work. This cycle is repeated for a predefined overall number of iterations.

In this application, we have 12 islands each evolving a 1-sized population for various numbers of iterations and one of them is the root that performs collecting the bests and providing the islands with relevant data to initiate their populations. The idea is presented in Figure 2.

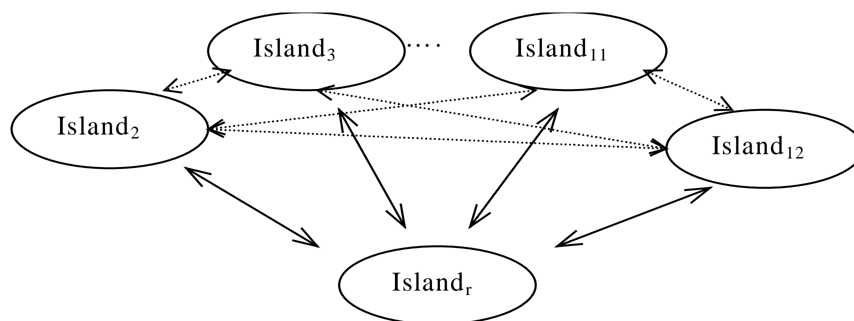

Figure 2; inter-islands relationships for parallel MSA

The islands communicate with one another by letting them exchange information as well as reporting their bests to the Root Island (Island$_r$ in Figure 2) at the end of every period. The experiments are launched on DRM by creating the Root Island first on the root node. The Root Island creates other islands and randomly dispatches them on randomly selected living nodes

---

[3] We will call agents as island anymore to be compatible to the glossary of DREAM Project.

of DRM by providing them with a population that formed of completely different solutions. The whole size of population operated on within an experiment is thus 12. By this application, we have got more chances for each particular solution as well as a more divers population, which provides different landscape of solutions to search on.

4. Experimental Results and Discussion

This experimental study has been done on a Pentium III 700MHz double processor computer, running Windows 2000. All the software are developed using Sun Java JDK1.3.1. We have developed two different implementations of simulated annealing to solve UFL problems as well as a parallel implementation on DRM environment. The simulated annealing implementations differ from each other by the size of population taken into account and depending on that the number of iterations that is set to run the algorithms. The DRM based parallel implementation of simulated annealing established on the idea of Aydin and Fogarty (2002), that is, spreading a big population of solutions over a number of identical parallel and asynchronous islands. The details of these works are coming in the following paragraphs. Beside these, in order to have a fair comparison with the latest work, we encoded a genetic algorithm that is done by Jaramillo et al (2002) to run on the same resources and using the same language (Java). In fact, the software has been encoded in FORTRAN 90 and executed on PC equipped by Intel Pentium running at 200MHz. Regarding the big difference between our conditions and the conditions they had, it would not be fair, if we just consider their results as standing in the paper. We encoded the algorithm in Java without missing any details given in the paper. The tackled problems are very well known benchmarks that are accessible on OR Library (Beasley, 1996). The benchmarks are named as indicated in the first column of each table. The first four benchmarks are 16 x 50, second four are 25 x 50, the third four are 50 x 50 and finally, the last three are 100 x 1000 sized problems as given in the second column of Table 1.

Table 1 presents the results gained by the genetic algorithm implemented by Jaramillo et al (2002). The columns show the information in the following order: the name of the benchmark, the size of the benchmark, optimum value, the average of results found, the arithmetic difference between the optimum and the average values, the percentage of that difference over optimum value and the CPU time taken for finding that average value. The quality of results we have found are slightly worse than those they have already given in their paper, this may be because of the random numbers used in the execution. The most important aspect is the time consumed for finding these solutions, which are much longer than that are given in the paper. The reason behind this is the time consuming of Java over the other languages.

Table 1 Experimental results obtained from GA done by Jaramillo et al (2002).

Benchmark	*Size (n x m)*	*Optimum*	*Average of Found Results*	*Difference*	*% Diff.*	*CPU Time in Seconds*
Cap 71	16 x 50	932615.75	932615.75	0.00	0.00000	0.287
Cap 72	16 x 50	977799.40	977799.40	0.00	0.00000	0.322
Cap 73	16 x 50	1010641.45	1010975.18	333.73	0.00033	0.773
Cap 74	16 x 50	1034976.98	1034976.98	0.01	0.00000	0.200
Cap 101	25 x 50	796648.44	796804.85	156.42	0.00020	0.801
Cap 102	25 x 50	854704.20	854704.20	0.00	0.00000	0.896
Cap 103	25 x 50	893782.11	893915.85	133.74	0.00015	1.371
Cap 104	25 x 50	928941.75	928941.75	0.00	0.00000	0.514
Cap 131	50 x 50	793439.56	793951.43	511.86	0.00065	6.663
Cap 132	50 x 50	851495.33	851495.32	0.01	0.00000	5.274
Cap 133	50 x 50	893076.71	893409.64	332.93	0.00037	7.189
Cap 134	50 x 50	928941.75	928941.75	0.00	0.00000	2.573
Cap A	100 x 1000	17156454.48	17156454.48	0.00	0.00000	184.422
Cap B	100 x 1000	12979071.58	13001435.08	22363.50	0.00172	510.445
Cap C	100 x 1000	11505594.33	11520610.16	15015.83	0.00131	591.516

Table 2 presents the results found from SA-I that is given in Figure 1 with a population of 5 individuals operated by that given algorithm for 300 times. As a result, the population has been operated 60000 times, since the algorithm iterates an individual for 200 times. The manipulation per individual is therefore 12000 times overall. The results are better than those found by GA with some deviations from the optimums in 4 of 15 benchmarks. We realised that these benchmarks are really tougher than the others are. Experiments are done 50 times for each benchmark. The CPU time consumed is measured as the time of the last best result found.

Table 2 Experimental results obtained from SA-I

Benchmark	*Optimum*	*Average of Found Results*	*Difference*	*%Dif*	*CPU Time in Seconds*
Cap 71	932615.75	932615.75	0.00	0.00000	0.041
Cap 72	977799.40	977799.40	0.00	0.00000	0.028
Cap 73	1010641.45	1010641.45	0.00	0.00000	0.031
Cap 74	1034976.98	1034976.98	0.00	0.00000	0.018
Cap 101	796648.44	796648.44	0.00	0.00000	0.256
Cap 102	854704.20	854704.20	0.00	0.00000	0.098
Cap 103	893782.11	893782.11	0.00	0.00000	0.119
Cap 104	928941.75	928941.75	0.00	0.00000	0.026
Cap 131	793439.56	793502.76	63.20	0.00008	2.506
Cap 132	851495.33	851495.33	0.00	0.00000	0.446
Cap 133	893076.71	893093.67	16.96	0.00002	0.443
Cap 134	928941.75	928941.75	0.00	0.00000	0.079
Cap A	17156454.48	17156454.48	0.00	0.00000	17.930
Cap B	12979071.58	12988199.17	9127.59	0.00070	91.937
Cap C	11505594.33	11519306.50	13712.17	0.00119	131.345

Table 3 shows the results obtained by SA II algorithm that is the same as the other except the size of population and the number of overall iterations. In this case, population size is extended

to 10 and the number of overall iterations have been increased to 2000 so that the number of total modification per individual is 40000. The results are much better in terms of quality of results, but not that much better in terms of the time consumed, although it is more or less the same in the case some benchmarks and worse in others. But, still benchmark C is not met for 100% as not met the optimum in few runs out of 50.

Table 3 Experimental results obtained from SA-II

Benchmark	*Optimum*	*Average of Found Results*	*Difference*	*% Dif.*	*CPU Time in Seconds*
Cap 71	932615.75	932615.75	0.00	0.00000	0.040
Cap 72	977799.40	977799.40	0.00	0.00000	0.020
Cap 73	1010641.45	1010641.45	0.00	0.00000	0.022
Cap 74	1034976.98	1034976.98	0.00	0.00000	0.013
Cap 101	796648.44	796648.44	0.00	0.00000	0.250
Cap 102	854704.20	854704.20	0.00	0.00000	0.085
Cap 103	893782.11	893782.11	0.00	0.00000	0.156
Cap 104	928941.75	928941.75	0.00	0.00000	0.031
Cap 131	793439.56	793439.56	0.00	0.00000	1.960
Cap 132	851495.33	851495.33	0.01	0.00000	0.828
Cap 133	893076.71	893076.71	0.00	0.00000	0.991
Cap 134	928941.75	928941.75	0.00	0.00000	0.111
Cap A	17156454.48	17156454.48	0.00	0.00000	29.699
Cap B	12979071.58	12979071.58	0.00	0.00000	98.563
Cap C	11505594.33	11506850.11	1255.78	0.00011	184.263

Table 4 Experimental results obtained from parallel SA on DRM

Benchmark	*Optimum*	*Average of Found Results*	*Difference*	*% Dif.*	*CPU Time in Seconds*
Cap 71	932615.75	932615.75	0.00	0.00000	0.013
Cap 72	977799.40	977799.40	0.00	0.00000	0.014
Cap 73	1010641.45	1010641.45	0.00	0.00000	0.011
Cap 74	1034976.98	1034976.98	0.00	0.00000	0.008
Cap 101	796648.44	796648.44	0.00	0.00000	0.046
Cap 102	854704.20	854704.20	0.00	0.00000	0.037
Cap 103	893782.11	893782.11	0.00	0.00000	0.115
Cap 104	928941.75	928941.75	0.00	0.00000	0.010
Cap 131	793439.56	793439.56	0.00	0.00000	0.207
Cap 132	851495.33	851495.33	0.01	0.00000	0.160
Cap 133	893076.71	893076.71	0.00	0.00000	0.103
Cap 134	928941.75	928941.75	0.00	0.00000	0.019
Cap A	17156454.48	17156454.48	0.00	0.00000	1.392
Cap B	12979071.58	12979071.58	0.00	0.00000	7.995
Cap C	11505594.33	11505594.33	0.00	0.00000	18.017

Table 4 contains the results gained by the parallel implementation of MSA algorithm that is given in Figure 1. Here, we developed our system to be able to run on DRM network in parallel. The system consists of 12 identical MSA islands working independently and communicate autonomously, where each island tackled a single individual, rather than a

population. The idea here is to spread a population of solutions over the distributed islands to have alternative operations in parallel. The results are very impressive and the best among those we examined so far. All benchmarks have been solved meeting the optimal values in a very short time comparing the other applications so far.

The results of four different applications are summarised on Table 5 showing the superiority of the parallel implemented MSA algorithm over the others. The main difficulty with the methods working with individuals is the more likely trap of sticking in local optimum depending on the initial solution. In the case of population, the diversity of solutions laying there lets the algorithm easily go through various paths towards optimum solution so that it accomplishes the mission easily with meeting the optimum solution. In the parallel MSA case, a divers initial population is spread over the islands to run all in parallel, and at the end, the best of found solutions in terms of both time and quality of the result is collected as the final best. So this gives the method more power than the others do.

The other MSA implementations have some controversial situation comparing to each other. SA-I is slightly better than SA-II with respect to CPU time consumed, but slightly worse in terms of quality of the results. In fact, SA-I has a lower size of population and number of iterations than SA-II. The larger the population, the longer the time needed to process. So, if more precise results are desired, it necessary to let the systems consuming more time.

Table 5 Summary of results gained from different algorithms for comparison

Benchmarks	*%Dif*				*CPU Time in sec*			
	GA	SA-I	SA-II	Parallel SA	GA	SA-I	SA-II	Parallel SA
Cap 71	0.00000	0.00000	0.00000	0.00000	0.287	0.041	0.040	0.013
Cap 72	0.00000	0.00000	0.00000	0.00000	0.322	0.028	0.020	0.014
Cap 73	0.00033	0.00000	0.00000	0.00000	0.773	0.031	0.022	0.011
Cap 74	0.00000	0.00000	0.00000	0.00000	0.200	0.018	0.013	0.008
Cap 101	0.00020	0.00000	0.00000	0.00000	0.801	0.256	0.250	0.046
Cap 102	0.00000	0.00000	0.00000	0.00000	0.896	0.098	0.085	0.037
Cap 103	0.00015	0.00000	0.00000	0.00000	1.371	0.119	0.156	0.115
Cap 104	0.00000	0.00000	0.00000	0.00000	0.514	0.026	0.031	0.010
Cap 131	0.00065	0.00008	0.00000	0.00000	6.663	2.506	1.960	0.207
Cap 132	0.00000	0.00000	0.00000	0.00000	5.274	0.446	0.828	0.160
Cap 133	0.00037	0.00002	0.00000	0.00000	7.189	0.443	0.991	0.103
Cap 134	0.00000	0.00000	0.00000	0.00000	2.573	0.079	0.111	0.019
Cap A	0.00000	0.00000	0.00000	0.00000	184.422	17.930	29.699	1.392
Cap B	0.00172	0.00070	0.00000	0.00000	510.445	91.937	98.563	7.995
Cap C	0.00131	0.00119	0.00011	0.00000	591.516	131.345	184.263	18.017

5. Conclusion

Facility layout problems have been studied for many years and thus there is a very rich literature in operations research for this kind of problems. Since UFL problems have NP-Hard nature, the larger the size of the problem, the harder to find the optimal solution and furthermore, the longer to reach a reasonable results. This paper discussed the examination of two new simulated annealing approaches for UFL problems with some useful comparisons with the latest genetic algorithm approach (Jaramillo et al., 2002) to the problem. As it is mentioned in many works, the simulated annealing approaches may be very successful in terms of quality of the solutions but not so impressive with respect to the CPU times. These approaches presented in this paper are seeking for the utilising of the power of the method in getting quality of solutions within shorter time. The evolutionary approach we used to improve simulated annealing yielded very good results compared to what the GA of Jaramillo et al. (2002) produced for both model we implemented.

Apart from all these, we have used distributed and parallel programming based multi agent implementations that gave much better results with respect to bot quality of solutions and CPU time consumed. This implementation is done using DRM, the network of virtual machines.

Acknowledgement

This work is funded as part of the European Commission Information Society Technologies Programme (Future and Emerging Technologies). The authors have sole responsibility for this work, it does not represent the opinion of the European Community, and the European Community is not responsible for any use that may be made of the data appearing herein.

References

1. Alves M.L. and Almeida, M.T. Simulated Annealing Algorithm for the Simple Plant Location Problem: A Computational Study. *Rev. Invest.* **12** (1992).
2. Aydin, M.E.,Fogarty, T.C., (2001), " Simulated annealing with evolutionary process for job-shop scheduling problems", In: EUROGEN 2001 - *Evolutionary Methods for Design, Optimisation and Control with Applications to Industrial Problems*, 19-21 September 2001, Athens, Greece
3. Aydin, M.E.,Fogarty, T.C., (2002), " A modular simulated annealing algorithm for multi-agent systems: A job shop scheduling application", *In Proc. of ICRM 2002 (2nd International Conference of Responsive Manufacturing),* 26-18 June 2002,Gaziantep, Turkey.
4. Beasley, J.E., (1993), "Lagrangean heuristics for location problems", *European J. Oper. Res.* **65** (1993) 383-399.

5. Beasley, J.E. Obtaining Test Problems *via* Internet. *J. Global Optim.* **8** (1996) 429-433, http://mscmga.ms.ic.ac.uk/info.html
6. Beasley, J.E., Chu, P.C., (1996), "A genetic algorithm for the set covering problem", *European J. Oper. Res.* **94** (1996) 392-404.
7. Conn A.R. and Cornuejols, G. A Projection Method for the Uncapacitated Facility Location Problem. *Math. Programming* **46** (1990) 273-298.
8. Dream Web Site, http://www.world-wide-dream.org
9. Erlenkotter, D. A Dual-Based Procedure for Uncapacitated Facility Location. *Oper. Res.***26** (1978) 992-1009.
10. Guignard, M. A Lagrangean Dual Ascent Algorithm for Simple Plant Location Problems, *European J. Oper. Res.* **35** (1988) 193-200.
11. Holmberg, K., (1995), *Experiments with Primal-Dual Decomposition and Subgradient Methods for the Uncapacitated Facility Location Problem*, Research Report LiTH-MAT/OPT-WP-1995- 08, Optimization. Department of Mathematics, Linkoping Institute of Technology, Sweden.
12. Holmberg, K., and Jörnsten, K., (1996), "Dual Search Procedures for The Exact Formulation of The Simple Plant Location Problem with Spatial Interaction", *Location Science* 4 (1996) 83 – 100
13. Jaramillo, J.H., Bhadury, J., Batta, R., (2002), "On the use of genetic algorithms to solve location problems", *Computers & Operations Research*, 29 (202), 761-779.
14. Jelasity M., Preuβ M., Peachter B., "A scalable and robust framework for distributed applications", CEC'02: The 2002 World Congress on Computational Intelligence, *12-17 May 2002:* Honolulu, HI, U.S.A.
15. Koerkel, M. (1989) On the Exact Solution of Large-Scale Simple Plant Location Problems. *European J. Oper. Res.* **39** (1989) 157-173.
16. Kratica J., Tošiæ D., Filipoviæ V., Ljubiæ I., (2001) "Solving the Simple Plant Location Problem by Genetic Algorithms", RAIRO - Operations Research, 35, No. 1, 127-142.
17. Peachter B., Back T., Schoenauer M., Sebag M., Eiben A. E., Merelo J. J., Fogarty T. C., (2000), " A distributed resource evolutionary algorithm machine (DREAM)" in Proc. of the Congress of Evolutionary Computation 2000 (CEC2000). IEEE, 2000, pp. 951-958, IEEE Press
18. Simao H.P. and Thizy, J.M., (1989), A Dual Simplex Algorithm for the Canonical Representation of the Uncapacitated Facility Location Problem. *Oper. Res. Lett*. **8** (1989) 279-286
19. Vaithyanathan, S., Burke, L. and Magent, M.A., (1996), Massively Parallel Analog Tabu Search Using Neural Networks Applied to Simple Plant Location Problem, *European J. Oper. Res.* 93 (1996) 317-330.