Available online at www.sciencedirect.com
SCIENCE DIRECT®
European Journal of Operational Research 157 (2004) 576–591
EUROPEAN JOURNAL OF OPERATIONAL RESEARCH
www.elsevier.com/locate/dsw

Discrete Optimization

A simple tabu search for warehouse location

Laurent Michel, Pascal Van Hentenryck *

Department of Computer Science, Brown University, Box 1910, Providence, RI 02912, USA

Received 11 June 2002; accepted 17 March 2003
Available online 23 September 2003

Abstract

The uncapacitated warehouse location problem (UWLP) is a heavily studied combinatorial optimization problem, which has been tackled by various approaches, such as linear-programming branch and bound, genetic algorithms, and local search. This paper presents an simple, yet robust and efficient, tabu-search algorithm for the UWLP. The algorithm was evaluated on the standard OR Library benchmarks and on the $\mathtt{M}^*$ instances which are very challenging for mathematical programming approaches. The benchmarks include instances of size 2000×2000. Despite its conceptual and programming simplicity, the algorithm finds optimal solutions to all benchmarks very quickly and with very high frequencies. It also compares favorably with state-of-the-art genetic algorithms and should be a very valuable addition to the repertoire of tools for uncapacitated warehouse location due its simplicity and effectiveness.

Keywords: Tabu search; Meta-heuristics; Warehouse location

1. Introduction

Given a set of n warehouses and a set of m stores, the uncapacitated warehouse location problem (UWLP) consists of choosing a subset of warehouses that minimizes the fixed costs of the warehouses and the transportation costs from the warehouses to the stores.

The UWLP has attracted considerable attention in mathematical programming. (See [10,13,22] for surveys of some of these approaches.) Many specific branch and bound algorithms were developed, including dual and primal–dual approaches [11,17]. Dual-based and primal–dual algorithms are very effective on the OR Library benchmarks for the UWLP [5]. However, they experience significant difficulties and exhibit exponential behaviour on the $\mathtt{M}^*$ instances [21]. These instances model real situations, have a large number of suboptimal solutions, and exhibit a strong tension between transportation and fixed costs, which makes it difficult to eliminate many warehouses early in the search.

Genetic algorithms have been shown to be very successful on the UWLP. In a series of papers spanning over several years (e.g., [12,19–21]), Kratica et al. have shown that genetic algorithms find optimal solutions on the OR Library and the $\mathtt{M}^*$ instances (whenever the optimal solutions are known) with high frequencies and very good efficiency. Their final algorithm uses clever

* Corresponding author. Tel.: +1-401-8637634; fax: +1-401-8637657.
E-mail address: pvh@cs.brown.edu (P. Van Hentenryck).

0377-2217/$ - see front matter
doi:10.1016/S0377-2217(03)00247-9

implementation techniques such as caching and bit vectors to avoid recomputing the objective function which is quite costly on large-scale problems. Ref. [21] also contains a detailed comparison with mathematical programming approaches and shows that the speed-up of the genetic algorithm over mathematical programming approaches increases exponentially with problem size on the $\mathtt{M}^*$ instances.

Various heuristic search algorithms have also been proposed but with less success. Ref. [2] presents simulated annealing algorithms which produce high-quality solutions but are quite expensive in computation times. Ref. [1] presents the only tabu-search algorithm we are aware of. The algorithm generates $5n$ neighbors at each iteration and moves to the best neighbor which is not tabu and improves the current value of the objective function. Each of these iterations takes significant computing time, which limits the applicability of the algorithm.

This paper originated as an attempt to find out whether there exists a tabu-search algorithm, which would be robust, efficient, and competitive with state-of-the-art genetic algorithms. It presents a very simple tabu-search algorithm which performs amazingly well on the UWLP. The algorithm uses a linear neighborhood and essentially takes $O(m \log n)$ time per iteration. It finds optimal solutions on the OR Library and the $\mathtt{M}^*$ instances (whenever the optimal solution is known) with high frequencies. It also outperforms the state-of-the-art genetic algorithm of Kratica et al., both in efficiency and robustness.

The main contributions of the paper can be summarized as follows:

1. It presents the first efficient and robust tabu-search algorithm which outperforms, or is competitive with, branch & bound and genetic algorithms in terms of solution quality, robustness, and efficiency.
2. The algorithm is extremely simple to understand and to implement, which makes it an appealing approach for practitioners interested in uncapacitated warehouse location.
3. The algorithm is easy to tune. It has a single parameter which controls the termination of the algorithm and is easy to tune in practice. The paper also describes in detail various tradeoffs between efficiency and solution quality obtained by varying this parameter.

As a consequence, we believe that the new tabu-search algorithm is a valuable addition to the repertoire of algorithms for the UWLPs. The rest of the paper is organized as follows. Section 2 defines the UWLP, Section 3 briefly describes prior work, and Section 4 presents the tabu-search algorithm. Section 5 reports the experimental results and Section 6 concludes the paper.

2. Uncapacitated warehouse location

We are given a set of n warehouses W and a set of m stores S. Each warehouse w has a fixed cost f_w and the transportation cost from warehouse w to store s is given by c_{ws}. The problem is to find a subset of warehouses and an assignment of warehouses to the stores to minimize the fixed and the transportation costs. Observe that, once the warehouses are selected, it suffices to assign the stores to their closest warehouse. As a consequence, the problem consists of finding a subset *Open* of warehouses minimizing the function

$$obj(Open) = \sum_{w \in Open} f_w + \sum_{s \in Stores} \min_{a \in Open} c_{as}.$$

3. Prior work

In mathematical programming, considerable attention has been devoted to the UWLP. It is beyond the scope of this paper to review the wealth of results in that area. See the excellent survey [8,13,22] for more information. One of the main results has been the development of linear-programming-based branch and bound algorithms. The standard reference in this area is the DUALOC algorithm of Erlenkotter [11], a branch and bound algorithm based on a dual descent. The algorithm performs very well on the instances of the OR Library and Ref. [22] indicates that "given its simplicity, speed, and availability, DUALOC may be the most efficient way to solve the (uncapacitated warehouse location) problem.'' Many papers have proposed improvements to DUALOC. For instance, Guignard [14] improves the relaxation by using Benders inequalities, Conn and Cornuéjols [8] proposed a projection also exploiting a dual formulation, and Holmberg [16] uses a primal-dual decomposition approach. Koerkel [17] proposes a primal-dual algorithm which shows significant improvements over DUALOC (speed-ups ranging from 10 to 100 are reported). Approximation algorithms for the uncapacitated warehouse location has also been studied heavily. See, for instance, [3,4,6,24]. These algorithms often use the linear-programming relaxation together with randomized rounding. Note that Ref. [3] shows that local minima of a local search algorithm using insertions, deletions, and swaps of warehouses have guaranteed quality performance in the worst-case. We will compare our tabu-search algorithm to DUALOC later in this paper to contrast its performance, both in terms of efficiency, solution quality, and robustness. Note that DUALOC is an exact algorithm: it is guaranteed to return the optimal solution upon completion.

Recent research has shown that genetic algorithms are excellent meta-heuristic methods for uncapacitated warehouse location. In a series of papers spanning over several years (e.g., [19–21]), Kratica et al. proposed increasingly sophisticated genetic algorithms. Ref. [21] is of special interest. It contains a comprehensive and excellent description of a genetic algorithm and its comparison to DUALOC. The algorithm uses a rank-based selector where about 1/3 of the population (150 individuals) is replaced at each iteration. The new individuals are generated using crossover and mutation operators. The algorithm uses several additional refinements (e.g., fitness adaption) as well as clever implementation techniques such as caching [18]. The algorithm was evaluated on the standard OR Library benchmarks and on the class of $\mathtt{M}^*$ instances proposed by the authors. These instances are extremely challenging for branch and bound algorithms: They contain many suboptimal solutions and very few useless facilities (contrary to the OR Library benchmarks). This is due to the great tension between transportation and fixed costs in these instances. As a consequence, branch and bound algorithms explore substantial portions of the search space and the duality gap is large. See [21] for a complete description of these instances. The genetic algorithm in [21] is shown to be very efficient and to produce optimal solutions (when they are known) with high frequencies. On the $\mathtt{M}^*$ instances, the gain performance is growing exponentially with the size of the problem compared to DUALOC. On the largest instances, DUALOC cannot complete execution in reasonable time and is far from the best solution returned by the genetic algorithm. These results are described in more detail later in the paper. Ref. [12] describes another interesting genetic algorithm where rank selection is replaced by fine-grained tournament selection. The results seem to indicate that tournament selection outperforms rank selection. Unfortunately, no solution quality and robustness results are given in the paper, which precludes direct comparison with that algorithm. Indeed, as we will show later in the paper, it is possible to reduce computation times drastically if we sacrifice quality or robustness slightly in our tabu-search algorithm.

Ref. [1] is the only tabu-search algorithm we are aware of. The main step of the algorithm generates $5n$ neighbors by flipping warehouses. The best such neighbor which is not tabu is selected as the next state if it improves the objective function. This main step is executed for a number of iterations. The algorithm also uses a sophisticated and effective heuristic as the starting point of the tabu search. The algorithm is evaluated on a subset of the OR Library benchmarks (but not on `capa`, `capb`, and `capc`, which are the most difficult). The paper indicates that optimal solutions were found by the algorithm, but do not report robustness results. As we will show later on, this algorithm is not competitive from an efficiency standpoint. Our tabu-search algorithm uses a much simpler neighborhood together with a simple diversification procedure in order to obtain substantial speed-ups. Finally, Ref. [2] presents simulated annealing algorithms which produce high-quality solutions but they are quite expensive in computation times.

4. The tabu-search algorithm

We now describe the tabu-search algorithm. Since the only combinatorial part is the selection of warehouses, it is natural to represent a state in the tabu search by a vector $y = \langle y_1, \dots, y_n \rangle$, where y_w is 1 if warehouse w is open and 0 otherwise. In the following, we use the notation

$$Open(y) = \{w \in N \mid y_w = 1\}$$

to represent the warehouses that are opened in a state y.

4.1. Neighborhood

The neighborhood is extremely simple. It consists of simply flipping the status of a warehouse. Given a state y, the neighborhood $\mathcal{N}(y)$ of y is defined as

$$\mathcal{N}(y) = \{flip(y, w) \mid w \in W\},$$

where

$$flip(\langle y_1, \dots, y_n \rangle, w) = \langle y_1, \dots, y_{w-1}, !y_w, y_{w+1}, \dots, y_n \rangle.$$

Obviously, $|\mathcal{N}(y)| = n$.

4.2. Tabu search

The tabu-search algorithm uses a tabu-list T which contains the set of warehouses that cannot be flipped. At each iteration, the algorithm considers the set of neighbors which are not tabu and whose gains are maximal. In other words, the set of non-tabu neighbors is defined as

$$\mathcal{N}^T(y) = \{flip(y, w) \mid w \in W \setminus T\}.$$

The best objective value of the neighbors is defined as

$$bestObj(y) = \max_{e \in \mathcal{N}^T(y)} obj(Open(e)),$$

and the set of neighbors considered by the tabu search is defined as

$$\mathcal{N}^*(y) = \{e \in \mathcal{N}^T(y) \mid obj(Open(e)) = bestObj(y)\}.$$

If the neighbors in $\mathcal{N}^*(y)$ do not degrade the current solution, the algorithm randomly moves to one of these neighbors by flipping, say, warehouse w. Warehouse w is then marked tabu for a number of iterations and the length of the tabu-list is updated. Otherwise, if the neighbors in $\mathcal{N}^*(y)$ degrade the current solution, the algorithm performs a diversification. It randomly selects an open warehouse and closes it.

This meta-heuristic is closely related to the general method advocated in [7] but it exploits specific problem knowledge in the diversification phase, which is critical to achieve adequate performance and robustness. The method in [7] randomly reinitializes x% for the variables (typically 5% or 10%) when stuck in a local minimum. We simply select an open warehouse and closes it, which is much more effective.

The algorithm can also be viewed as a degenerate form of variable-neighborhood search [15]. Variable-neighborhood search combines a local improvement procedure with a shaking component, which can be thought of as a diversification which becomes increasingly larger as computation proceeds. Our algorithm has only a very simple form of shaking, which only closes an open warehouse. However, additional diversification is provided through the tabu-list.

Fig. 1 describes the implementation of the tabu-search algorithm in more detail (including the parameters used in our implementation). The algorithm uses the set of best flips

$$bestFlips(y) = \{w \mid flip(y, w) \in \mathcal{N}^*(y)\}$$

and the best gain

$$bestGain(y) = obj(y) - bestObj(y)$$

to represent the neighbors and their objective value compactly. We will show how to maintain these values incrementally in the next two sections. The search iterates a number of steps. At each iteration, if the best gain is nonnegative, the algorithm randomly selects a warehouse in `bestFlips(y)` and flips its value. The tabu-list length is also adjusted using a standard scheme.

When the gain is negative, the algorithm randomly selects an open warehouse and closes it. The algorithm terminates when the objective function has not improved for 500 iterations. Note that the value 500 for `StabilityLimit`, the only parameter of the algorithm, was chosen to achieve the same robustness for solution quality as the

```
int nbStable = 0;
float best = obj;
S* = y;
int tLen = 10;
int stabilityLimit = 500;
while (nbStable < stabilityLimit) {
   float old = obj;
   if (bestGain(y) >= 0) {
      w = random(bestFlips(y));
      y_w = ! y_w;
      t[w] = it + tLen;
      if (obj < old && tLen > 2)
         tLen = tLen - 1;
      if (obj >= old && tLen < 10)
         tLen = tLen + 1;
      it = it + 1;
   } else {
      w = random(Open(y));
      y_w = 0;
   }
   if (obj < best) {
      best = obj;
      S* = y;
      nbStable = 0;
   } else
      nbStable = nbStable + 1;
}
```

Fig. 1. A tabu-search algorithm for uncapacitated warehouse location.

state-of-the-art genetic algorithm. This makes it possible to have meaningful performance comparison between the two algorithms. The experimental results discuss the impact of this parameter on the algorithm in great detail in the experimental section. The tabu-list is implemented by associating a counter `t[w]` with each warehouse `w`, by the integer `tLen` which keeps track of the length of the tabu-list, and (implicitly) by the iteration counter `it`. A warehouse is in the tabu-list if `t[w]>=it`. An assignment `t[w]=it+tLen` inserts a warehouse in the tabu-list for `tLen` iterations. The length of the tabu-list is adjusted by updating `tLen`. Note that we use a simple dynamic tabu-list, with typical initial and boundary values (which are used in many other applications). No effort has been spent trying to tune these values, since the algorithm performs extremely well with these typical values.

4.3. Data structures

To achieve good performance in practice, it is critical to maintain the best flips and the best gain incrementally during the search. As a consequence, our implementation maintains a collection of data structures incrementally. For each store s, the algorithm maintains the closest warehouse and the cost of the closest and the second closest warehouse, i.e.,

$$a_s = \arg\min_{w \in Open(y)} c_{ws}, \tag{1}$$

$$b_s = \min_{w \in Open(y)} c_{ws}, \tag{2}$$

$$d_s = \min_{w \in Open(y) \setminus \{w_{a_s}\}} c_{ws}. \tag{3}$$

The last two equations make it easy to compute the potential benefit of opening and closing a warehouse. The benefit g_w^- of closing warehouse w (assuming that w is opened) is given by

$$g_w^- = f_w - \sum_{s \in S_w} (d_s - b_s), \tag{4}$$

where S_w is the set of stores assigned to w, i.e.,

$$S_w = \{s \in S \mid a_s = w\}.$$

The benefit g_w^+ of opening warehouse w (assuming that w is closed) is given by

$$g_w^+ = -f_w + \sum_{s \in S} e_{ws}, \tag{5}$$

where

$$e_{ws} = \max(0, b_s - c_{ws}). \tag{6}$$

The gain g_w of flipping warehouse w can then be expressed as

$$g_w = \begin{cases} g_w^- & \iff & w \in Open(y), \\ g_w^+ & \iff & \text{otherwise.} \end{cases}$$

The best gain is given by

$$bestGain = \max_{w \in W \setminus T} g_w. \tag{7}$$

The best flips can now be defined as

$$bestFlips = \{w \in W \setminus T \mid g_w = bestGain\}. \qquad (8)$$

4.4. Incremental algorithms

We now discuss how Eqs. (1)–(8) can be maintained incrementally. In the following, we use a^0 to denote the value of a before the flip and a^1 to denote the value of a after the flip. Similar conventions are used for b and d.

4.4.1. Maintaining g^-

The values g^- are updated whenever values a_s, b_s and d_s are modified. More precisely, for every triplet $\langle a_s, b_s, d_s\rangle$ such that

$$a_s^1 \neq a_s^0 \vee b_s^0 \neq b_s^1 \vee d_s^0 \neq d_s^1,$$

the following update rules must be applied:

$$g^-_{a_s^0} = g^-_{a_s^0} + (b_s^0 - d_s^0),$$

$$g^-_{a_s^1} = g^-_{a_s^1} - (b_s^1 - d_s^1).$$

Observe that a_s^0 may, or may not, be equal to a_s^1.

4.4.2. Maintaining g^+

The values g^+ can also be maintained incrementally. From Eqs. (5) and (6), these values must only be updated when the value b_s changes, since c_{ws} is a constant. We now show which warehouses to consider when a store s has its value b_s updated.

Consider store s and its value b_s. The key observation is to notice that $\max(0, b_s - c_{ws})$ is a monotonically decreasing function for a permutation π_s of the vector c_{ws} (see Fig. 2). The permutation is simply obtained by sorting the warehouses by increasing order of the costs c_{ws}. As a consequence, it is easy to determine which warehouses are affected by the change in b_s and to update the values e_{ws} (and hence the values g_w^+) accordingly.

Consider first the case where $b_s^1 \leqslant b_s^0$, which corresponds to opening a warehouse. Fig. 3 depicts this situation. (The top solid curve denotes $\max(0, b_s^0 - c_{ws})$, while the bottom dashed curve denotes $\max(0, b_s^1 - c_{ws})$.) It is sufficient to update g_w^+ only for those w such that $c_{ws} \leqslant b_s^0$. The other warehouses are unaffected. We obtain the following update rules:

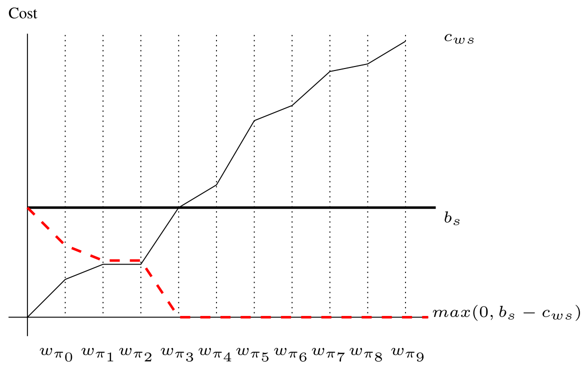

Fig. 2. Updating g^+.

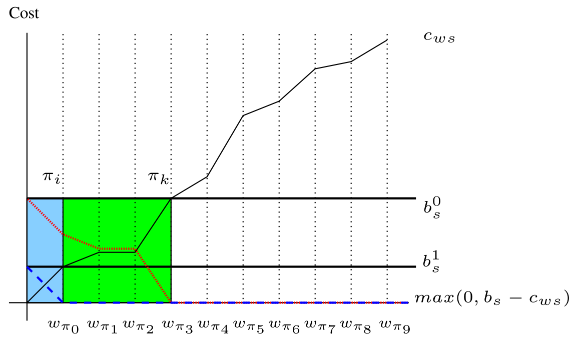

Fig. 3. Opening a warehouse.

$$g_w^+ = g_w^+ - (b_s^0 - b_s^1) \quad \forall\, w : c_{ws} < b_s^1,$$

$$g_w^+ = g_w^+ - (b_s^0 - c_{ws}) \quad \forall\, w : b_s^1 \leqslant c_{ws} < b_s^0.$$

Consider now the case where $b_s^1 \geqslant b_s^0$, which corresponds to closing a warehouse. Fig. 4 depicts this situation. (The bottom solid curve denotes

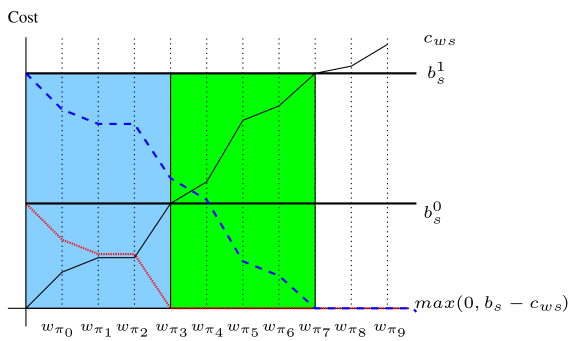

Fig. 4. Closing a warehouse.

$\max(0, b_s^0 - c_{ws})$, while the top dashed curve denotes $\max(0, b_s^1 - c_{ws})$.) It is sufficient to update g_w^+ only for those w such that $c_{ws} \leqslant b_s^1$. The other warehouses are unaffected. We obtain the following update rules:

$$g_w^+ = g_w^+ + (b_s^1 - b_s^0) \quad \forall\, w : c_{ws} < b_s^0,$$

$$g_w^+ = g_w^+ + (b_s^1 - c_{ws}) \quad \forall\, w : b_s^0 \leqslant c_{ws} < b_s^1.$$

4.4.3. Complexity analysis

We analyze the complexity of maintaining Eqs. (1)–(8). Note that the entries of the arrays a, b, and d of Eqs. (1)–(3) can be maintained with priority queues in $\mathrm{O}(m \log n)$. We use a typical implementation of priority queues based on a heap, i.e., a partially ordered tree (e.g., [9]). With this implementation, it is easy to maintain the two closest warehouses for a given customer. It suffices to insert in, or remove elements from, the priority queue, since the closest two warehouses can be found easily by looking at the top three nodes since, in a partially ordered tree, the root of any subtree has a lower priority than any of its children.

We now show that updating the entries in Eqs. (4)–(6) takes time linear in the number of updated values. This type of analysis was suggested in [23], since it captures the essence of many incremental algorithms maintaining an output under changes. It is particularly appropriate to analyze local search algorithms, since a move does not change the state much in general.

Updating g^- does not induce any asymptotic cost. Indeed, the update rule for each triplet takes $\Theta(1)$ time and each triplet $\langle a_s, b_s, d_s \rangle$ satisfies

$$a_s^1 \neq a_s^0 \vee b_s^0 \neq b_s^1 \vee d_s^0 \neq d_s^1,$$

which means that the update rule only adds a constant factor to the maintenance of the priority queues.

We now analyze the update of g^+. Define $\Delta(e)$ as the number of values e_{ws} which needs to be updated because of a flip. We show that the algorithm is linear in $\Delta(e)$, while the update of g^+ requires $\Omega(\Delta(e))$, yielding an optimal algorithm. Indeed, the value e_{ws} changes each time a warehouse w is considered by the above update rules for a store s. Hence the amount of work in the update rules is linear in the number of updated e_{ws}. In addition, updating g_w^+ when e_{ws} changes takes constant time. Note also that the update of g^+ is optimal: it is necessary to consider all the updated e_{ws}, since they cannot cancel out. This is due to the fact that either all the b_s increase (closing a warehouses) or all the b_s decrease (opening a warehouse).

Once Eqs. (1)–(6) are updated, Eqs. (7) and (8) can easily be computed in $\mathrm{O}(m \log m)$ time and $\mathrm{O}(m)$ time respectively. Overall, the maintenance of Eqs. (1)–(8) takes $\mathrm{O}(m \log n + \Delta(e))$ time, which is optimal. Note that, in practice, this cost is dominated by updating of the priority queues, as a profile of our implementation has shown.

5. Experimental results

This section describes the experimental results of the algorithm. Section 5.1 describes its efficiency, its solution quality, and its robustness with the value of its parameter `stabilityLimit` initialized to 500. As mentioned, this value was chosen to make the comparison with the genetic algorithm meaningful. Section 5.2 studies the robustness of the algorithm with respect to parameter `stabilityLimit`. Both the solution quality and the efficiency of the algorithm are studied. Sections 5.3–5.5 compare our algorithm with the state-of-the-art genetic algorithm, the dual-based branch and bound DUALOC, and the only tabu-search algorithm we are aware of. Section 5.6 summarizes the results.

In comparing our tabu-search algorithm with other approaches, we use the traditional approach of scaling the clock speed. In general, such scaling is conservative and favors the slower machines. In addition, our algorithm has low memory requirements (as can easily be seen from our data structures) and hence memory, and memory speed, is not an issue for our algorithm. These coarse comparisons are sufficient to show that our tabu-search algorithm can be implemented efficiently, that the constants are indeed very small, and that it is certainly at least competitive with previous algorithms, thus supporting the claims made in the introduction.

5.1. Performance of the algorithm

This section evaluates the performance of our tabu-search algorithm, both in terms of execution time and in terms of quality of the solution. The algorithm has a single parameter, `stabilityLimit`, which represents the number of iterations allowed without improvement of the best solution found so far. In the results given in this subsection, this parameter is set to 500. We discuss the impact of this parameter in the next section.

Table 1 depicts the experimental results on the standard OR Library benchmarks for uncapacitated warehouse location, as well as the `M*` instances from [21]. [1] Recall that the `M*` instances, which capture classes of real UWLPs [21], are very challenging for mathematical programming approaches because they have a large number of suboptimal solutions. Each benchmark was run 100 times on a Pentium IV 2 GHz running Debian Linux, since the starting point is entirely random.

The table reports the number of times the optimal solution was found (Opt), the best (BEST), average (AVG), and worst (WST) solutions found across the runs, and the average time in seconds to the best solution ($\mu(TS)$) and to complete the search ($\mu(TE)$). The last three columns report the standard deviation for the solution (expressed as a percentage for $\sigma(S)$), the time to the best solution, and the time to the completion of the execution. The `cap` benchmarks are from the OR Library. The other problems are the `M*` instances. Note that, on the larger `M*` instances, the optimal solutions are not known [21]. Optimal solutions are known on the `MO*`, `MP*`, `MQ*`, and the `MR1` and `MR2` instances. It is useful to note that the algorithm has no prior knowledge of the optimal solution, i.e., it cannot terminate early when the optimum solution is found.

As can be seen from Table 1, the algorithm is very robust. It finds optimal solutions with very high frequencies on all benchmarks. The variance on the quality of the solutions is always below 0.4% and is zero most of the time. In addition, the algorithm is extremely fast. It solves all the benchmarks in the OR Library in less than 2.5 seconds (total execution time) and finding the optimal solution after less than 0.9 seconds. In addition, it solves the largest 2000×2000 instances in about 40 seconds, while finding the optimal solutions in about 13 seconds. It should also be mentioned that there is considerable room for improvement in our implementation.

5.2. Robustness of the algorithm

As mentioned earlier, there is only one parameter in the algorithm: the number of iterations without improvement before completing the search (`stabilityLimit`). This subsection studies the impact of this parameter on the solution quality and on the performance of the algorithm.

5.2.1. Quality robustness

Table 2 gives the sensitivity of solution quality with respect to `stabilityLimit` and Fig. 5 depicts the results graphically. We evaluated the algorithm for various values of `stabilityLimit` between 16 and 500. For each value, we ran the algorithm 100 times on each of the benchmarks. The first six columns in Table 2 give the number of runs where the optimal value was found. The second set of six columns gives the variance (in percentage point) of the solution quality. In Fig. 5, the vertical axis represents the number of times the optimal solution was found, while the various values of `stabilityLimit` are displayed on the horizontal axis.

It can be seen that setting `stabilityLimit` = `250` gives very similar results to setting `stabilityLimit` = `500`. With `stabilityLimit` = `250`, the optimal solution is always found in more than 48% of the runs for all benchmarks. Moreover, the optimal solution is found in all the runs for most of the benchmarks. The quality results remain relatively good as we decrease `stabilityLimit` further. It is necessary to reduce `stabilityLimit` to 16 in order to obtain a variance exceeding 1% of the solution

[1] These instances were kindly given to us by J. Kratica.

Table 1
Experimental results for the local search algorithm

Bench	Size	Opt	BEST	AVG	WST	$\mu(TS)$	$\mu(TE)$	$\sigma(S)$ (%)	$\sigma(TS)$	$\sigma(TE)$
cap71	16×50	100	932 615.75	932 615.75	932 615.75	0.00	0.05	0.000	0.00	0.00
cap72	16×50	100	977 799.40	977 799.40	977 799.40	0.00	0.05	0.000	0.00	0.00
cap73	16×50	100	1 010 641.45	1 010 641.45	1 010 641.45	0.00	0.07	0.000	0.01	0.01
cap74	16×50	100	1 034 976.97	1 034 976.97	1 034 976.97	0.00	0.07	0.000	0.00	0.00
cap101	25×50	80	796 648.44	796 820.49	797 508.72	0.01	0.07	0.043	0.02	0.02
cap102	25×50	100	854 704.20	854 704.20	854 704.20	0.00	0.06	0.000	0.00	0.01
cap103	25×50	94	893 782.11	893 795.67	894 008.14	0.02	0.08	0.006	0.02	0.02
cap104	25×50	100	928 941.75	928 941.75	928 941.75	0.00	0.08	0.000	0.00	0.01
cap131	50×50	84	793 439.56	793 577.21	794 299.85	0.03	0.10	0.040	0.02	0.02
cap132	50×50	100	851 495.32	851 495.33	851 495.32	0.01	0.09	0.000	0.01	0.01
cap133	50×50	96	893 076.71	893 104.93	893 782.11	0.03	0.12	0.015	0.03	0.03
cap134	50×50	100	928 941.75	928 941.75	928 941.75	0.01	0.13	0.000	0.00	0.01
capa	100×1000	100	17 156 454.4	17 156 454.4	17 156 454.4	0.20	2.31	0.000	0.06	0.06
capb	100×1000	53	12 979 071.5	13 022 893.3	13 214 718.1	0.39	1.98	0.393	0.32	0.32
capc	100×1000	68	11 505 594.3	11 514 330.7	11 672 443.3	0.81	2.27	0.181	0.65	0.65
MO1	100×100	95	1156.91	1156.95	1157.70	0.10	0.50	0.015	0.08	0.09
MO2	100×100	100	1227.67	1227.67	1227.67	0.04	0.42	0.000	0.01	0.02
MO3	100×100	87	1286.37	1286.60	1288.18	0.14	0.57	0.047	0.11	0.11
MO4	100×100	100	1177.88	1177.88	1177.88	0.06	0.46	0.000	0.04	0.04
MO5	100×100	100	1147.60	1147.60	1147.60	0.05	0.46	0.000	0.04	0.05
MP1	200×200	100	2460.10	2460.10	2460.10	0.14	1.09	0.000	0.03	0.05
MP2	200×200	100	2419.32	2419.33	2419.32	0.14	1.15	0.000	0.03	0.05
MP3	200×200	100	2498.15	2498.15	2498.15	0.17	1.20	0.000	0.07	0.07
MP4	200×200	100	2633.56	2633.56	2633.56	0.15	1.18	0.000	0.04	0.05
MP5	200×200	100	2290.16	2290.16	2290.16	0.13	1.08	0.000	0.01	0.02
MQ1	300×300	100	3591.27	3591.27	3591.27	0.28	1.86	0.000	0.03	0.05
MQ2	300×300	100	3543.66	3543.66	3543.66	0.29	1.98	0.000	0.04	0.05
MQ3	300×300	100	3476.81	3476.81	3476.81	0.31	1.97	0.000	0.05	0.05
MQ4	300×300	100	3742.47	3742.47	3742.47	0.32	1.95	0.000	0.05	0.09
MQ5	300×300	100	3751.33	3751.33	3751.33	0.29	1.99	0.000	0.05	0.06
MR1	500×500	100	2349.86	2349.86	2349.86	0.85	3.91	0.000	0.26	0.26
MR2	500×500	100	2344.76	2344.76	2344.76	0.77	3.84	0.000	0.07	0.07
MR3	500×500	100	2183.24	2183.23	2183.24	0.88	3.80	0.000	0.23	0.23
MR4	500×500	100	2433.11	2433.11	2433.11	0.80	3.99	0.000	0.11	0.10
MR5	500×500	100	2344.35	2344.35	2344.35	0.79	3.98	0.000	0.07	0.08
MS1	1000×1000	100	4378.63	4378.63	4378.63	2.99	10.59	0.000	0.16	0.19
MS2	1000×1000	100	4658.35	4658.35	4658.35	3.21	11.70	0.000	0.31	0.47
MS3	1000×1000	100	4659.16	4659.16	4659.16	3.12	11.26	0.000	0.29	0.42
MS4	1000×1000	100	4536.00	4536.00	4536.00	4.27	12.76	0.000	1.77	2.05
MS5	1000×1000	100	4888.91	4888.91	4888.91	3.55	12.21	0.000	0.83	1.15
MT1	2000×2000	100	9176.51	9176.51	9176.51	12.97	39.30	0.000	2.45	6.50
MT2	2000×2000	100	9618.85	9618.85	9618.85	12.61	38.31	0.000	1.85	6.27
MT3	2000×2000	100	8781.11	8781.11	8781.11	12.31	34.87	0.000	1.46	3.56
MT4	2000×2000	100	9225.49	9225.49	9225.49	13.25	39.47	0.000	3.35	8.77
MT5	2000×2000	100	9540.67	9540.67	9540.67	12.99	40.13	0.000	2.92	9.59

quality. This means that, even for a `stabilityLimit` of 16, the vast majority of the results are still close to the optimal.

Table 3 gives the sensitivity of execution time with respect to `stabilityLimit` and Fig. 6 depicts the results graphically. Once again, we evaluated the algorithm for various values of `stabilityLimit` between 16 and 500. For each value, we ran the algorithm 100 times on each of the benchmarks. The first six columns in Table 2 give the execution time in seconds for various values of `stabilityLimit`. The second set of

Table 2
Sensitivity of the solution quality on parameter `stabilityLimit`

Bench	16	32	64	125	250	500	$\sigma(S)_{16}$ (%)	$\sigma(S)_{32}$ (%)	$\sigma(S)_{64}$ (%)	$\sigma(S)_{125}$ (%)	$\sigma(S)_{250}$ (%)	$\sigma(S)_{500}$ (%)
cap71	86	92	100	100	100	100	0.059	0.046	0.000	0.000	0.000	0.000
cap72	86	98	99	100	100	100	0.163	0.055	0.039	0.000	0.000	0.000
cap73	76	84	96	100	100	100	0.100	0.073	0.036	0.000	0.000	0.000
cap74	87	97	100	100	100	100	0.089	0.045	0.000	0.000	0.000	0.000
cap101	37	36	43	52	66	80	0.147	0.123	0.072	0.054	0.051	0.043
cap102	63	87	96	100	100	100	0.194	0.090	0.043	0.000	0.000	0.000
cap103	31	35	45	60	83	94	0.138	0.090	0.062	0.034	0.014	0.006
cap104	88	99	100	100	100	100	0.197	0.060	0.000	0.000	0.000	0.000
cap131	15	29	44	60	71	84	0.295	0.248	0.122	0.086	0.049	0.040
cap132	54	76	94	99	100	100	0.259	0.167	0.068	0.008	0.000	0.000
cap133	24	33	55	68	84	96	0.156	0.130	0.056	0.047	0.031	0.015
cap134	77	97	100	100	100	100	0.224	0.066	0.000	0.000	0.000	0.000
capa	73	90	100	100	100	100	0.772	0.612	0.000	0.000	0.000	0.000
capb	16	49	56	60	58	53	0.892	0.716	0.555	0.415	0.335	0.393
capc	4	8	19	18	48	68	1.013	0.569	0.405	0.271	0.166	0.181
capmo1	36	42	61	76	85	95	0.389	0.279	0.147	0.053	0.029	0.015
capmo2	95	100	100	100	100	100	0.160	0.000	0.000	0.000	0.000	0.000
capmo3	19	17	41	48	70	87	0.248	0.239	0.181	0.106	0.065	0.047
capmo4	62	73	87	97	100	100	0.490	0.340	0.216	0.102	0.000	0.000
capmo5	62	72	86	95	100	100	0.404	0.238	0.173	0.069	0.000	0.000
capmp1	85	97	100	100	100	100	0.210	0.122	0.000	0.000	0.000	0.000
capmp2	91	97	98	100	100	100	0.310	0.113	0.078	0.000	0.000	0.000
capmp3	59	71	84	99	100	100	0.397	0.263	0.220	0.041	0.000	0.000
capmp4	85	93	99	100	100	100	0.536	0.382	0.160	0.000	0.000	0.000
capmp5	98	100	100	100	100	100	0.258	0.000	0.000	0.000	0.000	0.000
capmq1	94	97	100	100	100	100	0.132	0.069	0.000	0.000	0.000	0.000
capmq2	84	98	99	100	100	100	0.135	0.005	0.004	0.000	0.000	0.000
capmq3	79	95	100	100	100	100	0.284	0.136	0.000	0.000	0.000	0.000
capmq4	72	92	100	100	100	100	0.117	0.054	0.000	0.000	0.000	0.000
capmq5	83	96	98	100	100	100	0.305	0.154	0.110	0.000	0.000	0.000
capmr1	67	79	87	96	100	100	0.507	0.296	0.245	0.143	0.000	0.000
capmr2	87	99	99	100	100	100	0.357	0.045	0.142	0.000	0.000	0.000
capmr3	61	85	94	96	100	100	0.437	0.297	0.183	0.095	0.000	0.000
capmr4	87	97	100	100	100	100	0.170	0.047	0.000	0.000	0.000	0.000
capmr5	89	97	100	100	100	100	0.248	0.111	0.000	0.000	0.000	0.000
capms1	92	97	100	100	100	100	0.025	0.016	0.000	0.000	0.000	0.000
capms2	85	99	100	100	100	100	0.047	0.005	0.000	0.000	0.000	0.000
capms3	82	94	100	100	100	100	0.372	0.166	0.000	0.000	0.000	0.000
capms4	56	64	74	86	94	100	0.259	0.238	0.198	0.157	0.110	0.000
capms5	62	73	92	95	100	100	0.169	0.091	0.037	0.030	0.000	0.000
capmt1	90	100	100	100	100	100	0.228	0.000	0.000	0.000	0.000	0.000
capmt2	73	91	100	100	100	100	0.111	0.049	0.000	0.000	0.000	0.000
capmt3	93	99	100	100	100	100	0.033	0.013	0.000	0.000	0.000	0.000
capmt4	86	99	100	100	100	100	0.206	0.004	0.000	0.000	0.000	0.000
capmt5	99	100	100	100	100	100	0.023	0.000	0.000	0.000	0.000	0.000

six columns gives the variance (in percentage point) of the execution time. In Fig. 6, the vertical axis represents the execution time in seconds, while the various values of `stabilityLimit` are displayed on the horizontal axis.

The results are particularly interesting. The first important observation is that the running time is a sublinear function of the number of stable iterations (for the various values of `stabilityLimit` we tried). When the number of stable iterations is

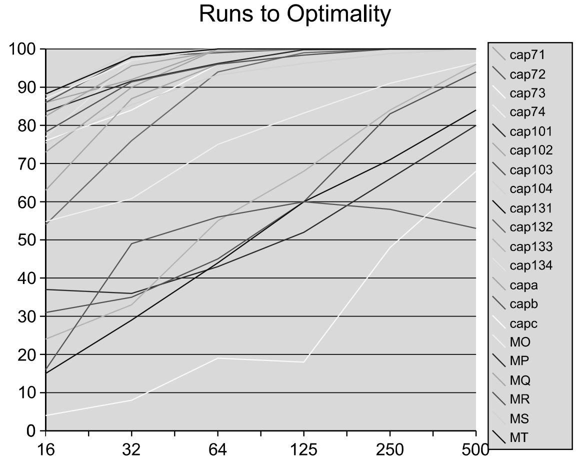

Fig. 5. Quality robustness wrt `stabilityLimit`. (The x-axis shows the values of `stabilityLimit`, while the y-axis shows the percentage of runs producing the optimal (or best-known) solution.)

multiplied by 2^5, the execution times only increase by a factor of about 4. This indicates that the algorithm spends most of its time finding the "optimal" solution, not detecting stability. The second important conclusion is that going above 500 stable iterations does not make much sense (at least for this collection of benchmarks). Indeed, running twice the algorithm with $\texttt{stabilityLimit} = 250$ will likely produce more robust results than one run with $\texttt{stabilityLimit} = 500$. For instance, on benchmark `capc`, the probability to return the optimal solution is about 73% for two runs with $\texttt{stabilityLimit} = 250$ and the computation time is roughly the same as one run with $\texttt{stabilityLimit} = 250$. As mentioned earlier in this paper, the setting $\texttt{stabilityLimit} = 500$ was chosen because it gives very similar robustness to the state-of-the-art genetic algorithm of [21], which makes it possible to have meaningful performance comparisons.

5.3. Comparison with the state-of-the-art genetic algorithm

In this subsection, we compare our algorithm (STS) with the state-of-the-art genetic algorithm of [21] which is probably the most successful algorithm available at this point. The comparison is given in Table 4, which is constructed using the results given in [21]. These results contains average execution times and robustness results for the various classes of benchmarks. As mentioned previously, STS was tested on the same instances as the state-of-the-art genetic algorithm of [21]. The first column describes the benchmark classes and the second column gives the number of instances in each class multiplied by the number of runs of each instance. The next three columns give the number runs producing the optimal solutions, the percentage of optimal solutions, and the execution in seconds. These columns are taken directly from [21]. The column $S(\mu(T))$ gives the scaled execution time using the clock speed of the machines used in the experiments. The ratio $2000/133$ was used to scale the running times. In general, this gives a conservative scaling which favors slower machines. The last two columns give the results for our STS algorithm by aggregating the results given previously. It is interesting to point out that the genetic algorithm in [21] uses different settings for the OR Library and the $\mathtt{M}^*$ instances. Our STS algorithm uses the same settings on all instances.

The experimental results are very interesting again. In general, the robustness of both algorithms is very similar: STS is slightly more robust in general, except on the cap81–104 benchmarks where the genetic algorithm is slightly better. As far as efficiency is concerned, STS is almost always faster and the difference seems to increase with the size of the benchmarks. On the largest MT instances, STS is more than three times as fast as the genetic algorithm. Also, the growth in time between the last two classes is 4.77 for the genetic algorithm and 3.28 for our algorithm. As a consequence, our STS algorithm seems to be a nice addition to the repertoire of algorithms for this problem. It seems to be the fastest algorithm available at this point, while achieving similar robustness results. Moreover, the algorithm is easy to tune, since it depends on a single parameter.

5.4. Comparison with a tabu-search algorithm

We now compare our algorithm with the tabu-search algorithm of [1]. This paper describes the

Table 3
Sensitivity of execution time on parameter `stabilityLimit`

Bench	16	32	64	125	250	500	σ_{16}	σ_{32}	σ_{64}	σ_{125}	σ_{250}	σ_{500}
cap71	0.00	0.01	0.01	0.01	0.03	0.05	0.00	0.00	0.00	0.00	0.00	0.00
cap72	0.00	0.01	0.01	0.02	0.03	0.05	0.00	0.00	0.00	0.00	0.00	0.00
cap73	0.00	0.01	0.01	0.02	0.03	0.07	0.00	0.00	0.00	0.00	0.00	0.01
cap74	0.00	0.01	0.01	0.02	0.04	0.07	0.00	0.00	0.00	0.00	0.00	0.00
cap101	0.00	0.01	0.01	0.02	0.04	0.07	0.00	0.00	0.00	0.00	0.01	0.02
cap102	0.00	0.01	0.01	0.02	0.03	0.06	0.00	0.00	0.00	0.00	0.00	0.01
cap103	0.00	0.01	0.01	0.02	0.05	0.08	0.00	0.00	0.00	0.01	0.01	0.02
cap104	0.01	0.01	0.01	0.02	0.04	0.08	0.00	0.00	0.00	0.00	0.00	0.01
cap131	0.01	0.01	0.02	0.03	0.06	0.10	0.00	0.01	0.01	0.01	0.01	0.02
cap132	0.01	0.01	0.02	0.03	0.05	0.09	0.01	0.00	0.00	0.01	0.01	0.01
cap133	0.01	0.01	0.02	0.04	0.07	0.12	0.01	0.01	0.01	0.01	0.02	0.03
cap134	0.01	0.02	0.03	0.04	0.07	0.13	0.00	0.00	0.00	0.01	0.01	0.01
capa	0.24	0.34	0.46	0.73	1.30	2.31	0.04	0.05	0.06	0.07	0.09	0.06
capb	0.20	0.30	0.43	0.67	1.12	1.98	0.04	0.06	0.08	0.11	0.20	0.32
capc	0.20	0.29	0.45	0.70	1.20	2.27	0.04	0.06	0.10	0.16	0.28	0.65
capmo1	0.05	0.07	0.10	0.16	0.29	0.50	0.01	0.01	0.02	0.03	0.05	0.09
capmo2	0.05	0.06	0.09	0.13	0.23	0.42	0.01	0.01	0.01	0.01	0.01	0.02
capmo3	0.05	0.07	0.10	0.17	0.32	0.57	0.01	0.01	0.02	0.04	0.07	0.11
capmo4	0.05	0.07	0.09	0.15	0.25	0.46	0.01	0.01	0.01	0.02	0.03	0.04
capmo5	0.05	0.07	0.09	0.15	0.26	0.46	0.01	0.01	0.01	0.03	0.04	0.05
capmp1	0.17	0.20	0.26	0.38	0.63	1.09	0.02	0.02	0.03	0.02	0.02	0.05
capmp2	0.17	0.20	0.27	0.39	0.65	1.15	0.01	0.02	0.02	0.02	0.02	0.05
capmp3	0.17	0.21	0.28	0.43	0.69	1.20	0.02	0.02	0.04	0.05	0.06	0.07
capmp4	0.17	0.22	0.27	0.41	0.67	1.18	0.02	0.02	0.03	0.03	0.03	0.05
capmp5	0.17	0.20	0.25	0.37	0.62	1.08	0.01	0.02	0.01	0.01	0.02	0.02
capmq1	0.34	0.40	0.49	0.69	1.11	1.86	0.02	0.03	0.04	0.04	0.04	0.05
capmq2	0.35	0.42	0.52	0.73	1.15	1.98	0.02	0.03	0.04	0.04	0.05	0.05
capmq3	0.36	0.42	0.53	0.73	1.19	1.97	0.03	0.04	0.06	0.04	0.06	0.05
capmq4	0.36	0.42	0.52	0.70	1.12	1.95	0.02	0.04	0.05	0.05	0.06	0.09
capmq5	0.35	0.41	0.52	0.72	1.18	1.99	0.03	0.03	0.05	0.06	0.08	0.06
capmr1	0.88	0.99	1.20	1.69	2.51	3.91	0.06	0.06	0.10	0.17	0.29	0.26
capmr2	0.87	1.00	1.19	1.60	2.39	3.84	0.05	0.07	0.08	0.09	0.08	0.07
capmr3	0.85	1.00	1.19	1.58	2.34	3.80	0.05	0.07	0.12	0.12	0.20	0.23
capmr4	0.87	1.01	1.20	1.66	2.40	3.99	0.05	0.07	0.08	0.09	0.09	0.10
capmr5	0.87	1.00	1.20	1.63	2.42	3.98	0.05	0.07	0.07	0.07	0.07	0.08
capms1	3.14	3.80	4.09	5.00	6.73	10.59	0.11	0.43	0.23	0.18	0.15	0.19
capms2	3.24	3.64	4.07	5.18	7.12	11.70	0.16	0.22	0.21	0.26	0.25	0.47
capms3	3.46	3.89	4.20	5.38	7.14	11.26	0.20	0.30	0.24	0.30	0.25	0.42
capms4	3.30	3.57	4.32	5.66	7.78	12.76	0.16	0.21	0.39	0.79	1.17	2.05
capms5	3.34	3.60	4.32	5.55	7.57	12.21	0.21	0.20	0.31	0.54	0.67	1.15
capmt1	12.64	13.84	15.31	18.34	24.32	39.30	0.44	1.40	0.44	0.82	0.98	6.50
capmt2	12.87	16.05	15.26	18.55	24.43	38.31	0.54	4.51	0.70	0.80	1.00	6.27
capmt3	12.39	13.83	14.66	17.46	22.49	34.87	0.44	2.64	0.58	0.67	0.60	3.56
capmt4	13.07	15.92	15.39	18.69	24.15	39.47	0.76	3.96	0.64	1.16	0.68	8.77
capmt5	12.69	14.22	15.05	18.14	23.77	40.13	0.44	1.10	0.69	0.62	0.46	9.59

only tabu-search algorithm we are aware of and it was the starting point of this research. The algorithm iterates a main step which consists in generating $5n$ neighbors and choosing the best non-tabu moves among them. It also uses an effective heuristic as the starting point of the tabu search. Table 5 reports the experimental results of this tabu-search algorithm as given in [1]. The third and fourth columns describes the quality (percentage compared to the optimum) and the

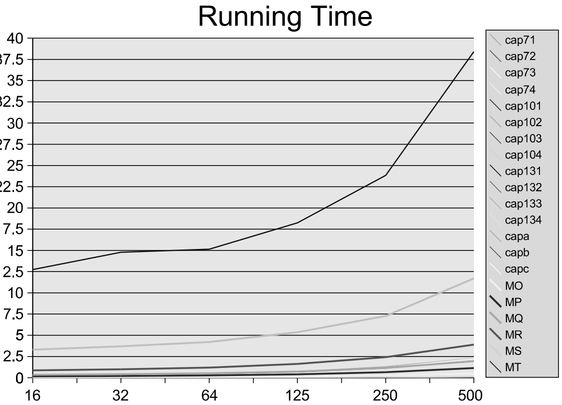

Fig. 6. Time robustness wrt `stabilityLimit`. (The x-axis shows the values of `stabilityLimit`, while the y-axis shows the computing time in seconds.)

time taken by their heuristics on a 48 633 MHz PC. The next two columns describe the quality and the time taken by their tabu-search algorithm, again on a 48 633 MHz PC. It is not clear how many experiments were run and what the robustness of the algorithm is, since this was not discussed in the paper. The last column gives the execution times when scaled to a Pentium IV 2 GHz. The ratio $2000/33$ was used to scale the running times, which gives a conservative estimate of the speed difference between the two machines. Only the results on a subset of OR Library benchmarks were given in [1]. As can be seen, these results are many orders of magnitude slower than our algorithm, since STS never takes more than 0.03 in average on these benchmarks. STS is both a simpler and much more effective algorithm.

Table 4
Comparison with the genetic algorithm from [21]

Bench	Genetic					STS	
	Runs	Opt	μ(%Opt)	$\mu(T)$	$S(\mu(T))$	μ(%Opt)	$S(\mu(T))$
cap41–74	$13 \times 20 = 260$	260	100.00	0.86	0.06	100.00	0.06
cap81–104	$12 \times 20 = 240$	240	100.00	1.26	0.08	93.50	0.07
cap111–134	$12 \times 20 = 240$	198	82.50	2.97	0.20	95.00	0.11
A–C	$3 \times 20 = 60$	42	70.0	83.10	5.52	73.67	2.19
MO	$5 \times 20 = 100$	93	93.0	5.49	0.36	96.40	0.48
MP	$5 \times 20 = 100$	100	100.0	16.60	1.10	100.00	1.14
MQ	$5 \times 20 = 100$	100	100.0	34.97	2.33	100.00	1.95
MR	$5 \times 20 = 100$	99	99.0	93.69	6.23	100.00	3.90
MS	$5 \times 20 = 100$	100	100.0	379.6	25.24	100.00	11.70
MT	$5 \times 20 = 100$	100	100.0	1812.3	120.51	100.00	38.41

Table 5
Comparison with the tabu search of [1]

Bench	Size	NBH%	NBH-T	UFLTSA%	UFLTSA-T	S(UFLTSA-T)
cap71	16×50	0.67	1.92	0	97.6	1.61
cap72	16×50	0.64	2.47	0	158.72	2.62
cap73	16×50	0.94	3.08	0	116.43	1.92
cap74	16×50	0.37	5.33	0	137.48	2.27
cap101	25×50	1.72	3.96	0	352.83	5.82
cap102	25×50	1.33	5	0	333.01	5.49
cap103	25×50	1.45	5.4	0	343.44	5.67
cap104	25×50	0.61	6.52	0	314.6	5.19
cap131	50×50	2.44	15.8	0	1548.72	5.55
cap132	50×50	1.99	16.21	0	1739.81	28.71
cap133	50×50	1.63	32.85	0	1684.22	27.79
cap134	50×50	0.61	36.85	0	1922.19	31.72

5.5. Comparison with LP-based branch and bound algorithms

We now compare our results with DUALOC [11], a dual-based branch and bound procedure which is the standard reference for the uncapacitated warehouse location. Of course, comparisons between heuristic and complete algorithms are always difficult, since the two classes of algorithms are very different in nature, but they give interesting information about these algorithms. Table 6 reports the results, most of which are taken from [21]. The second and third column report the computation times on a 133 MHz PC and when scaled to the speed of our machine. The fourth column describes the distance to the best known solution when DUALOC cannot complete in reasonable time. The last column gives the results of STS. It can be seen than DUALOC is particularly effective on the OR Library benchmarks. However, its performance degrades substantially on the `M*` instances. It becomes already about 100 slower than STS on the `MQ` instances. Moreover, it experiences much difficulty in finding optimal solutions on the `MR*` instances. Both `MR1` and `MR2` could not be solved in reasonable time and the distance with respect to the best known solution is quite significant. The `MS` and `MT` instances seem to be out of scope for DUALOC. Note that, on all the benchmarks where the optimal solution is known, STS was able to find it with very high frequencies, as reported earlier. Once again, it is natural to conclude that STS is a very effective algorithm for warehouse location, due to its computational efficiency, robustness, and simplicity.

5.6. Summary

Figs. 7 and 8 summarize the experimental results. They indicate that STS is a particularly

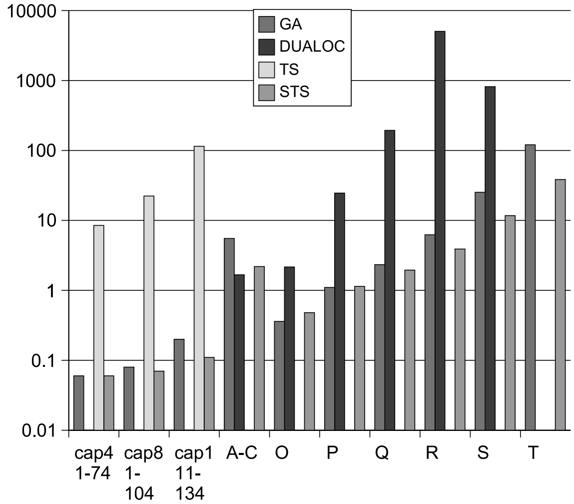

Fig. 7. Efficiency comparison of the various algorithms. (The *x*-axis gives the various classes of benchmarks, while the *y*-axis describes the scaled computation times in seconds.)

Table 6
Comparison with DUALOC [11]

Bench	Dualoc			STS
	DUALOC	*S*(DUALOC)	%*BS*	
cap41–74	**<0.01**	**<0.01**		0.06
cap81–104	**<0.01**	**<0.01**		0.07
cap111–134	**<0.01**	**<0.01**		0.11
A–C	**25.17**	**1.67**		2.19
MO	**32.54**	**2.16**		0.48
MP	**369.7**	**24.59**		1.14
MQ	**2913.7**	**193.76**		1.95
MR1	99 424	6611.7	9.4	3.91
MR2	68 008	4522.5	6.7	3.84
MR3	**54 167**	**3602.1**		3.8
MR4	**79 779**	**5305.3**		3.99
MR5	**78 444**	**5216.5**		3.98
MS	12 279	816.5	11.32	11.70
MT	NA	NA		38.41

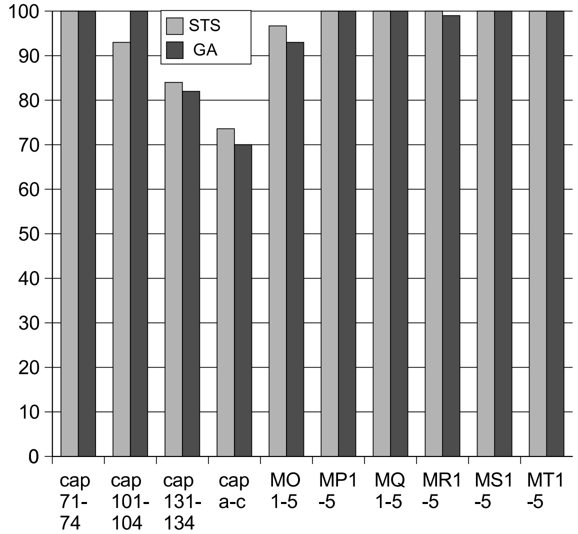

Fig. 8. Robustness comparison between the genetic algorithm and STS. (The x-axis gives the various classes of benchmarks, while the y-axis gives the percentage of runs where the optimal (or best-known) solution is found.)

effective algorithm for uncapacitated warehouse location. In general, it is faster than the other algorithms used in the comparison and it produces optimal results with very high frequencies. The state-of-the-art genetic algorithm is close in terms of performance (although the difference in efficiency seems to increase with size) and in robustness. DUALOC is very effective on the OR Library benchmarks but has significant difficulties on the M* instances. The tabu-search algorithm in [1] cannot compete with these algorithms in terms of efficiency.

6. Conclusion

The uncapacitated warehouse location problem (UWLP) has been studied heavily in combinatorial optimization, leading to excellent mathematical programming and genetic algorithms. This paper originated in an attempt to find out whether it was possible to design a tabu-search algorithm, which would be robust, efficient, and competitive with state-of-the-art genetic algorithms. It presented a very simple tabu-search algorithm which performs amazingly well on the UWLP. The algorithm uses a linear neighborhood, which flips a single warehouse at each iteration in essentially $O(m \log n)$ time. The algorithm finds optimal solutions on the OR Library and the M* instances (whenever the optimal solution is known) with high frequencies. It also outperforms, or is comparable to, the state-of-the-art genetic algorithm of Kratica et al., both in efficiency and robustness. Because of its effectiveness and its simplicity, we believe that the algorithm is a valuable addition to the repertoire of algorithms for the uncapacitated warehouse location.

Acknowledgements

Many thanks to Russell Bent for interesting discussions in this paper, to J. Kratica for kindly providing us with his hard M* instances, and to the reviewers for their comments which help improve the presentation of the results significantly. Pascal Van Hentenryck is partially supported by NSF ITR Award DMI-0121495.

References

[1] K.S. Al-Sultan, M.A. Al-Fawzan, A tabu search approach to the uncapacitated facility location problem, Annals of Operations Research 86 (1999) 91–103.
[2] M.L. Alves, M.T. Almeida, Simulated annealing algorithm for simple plant location problems, Revista Investigacao Operacional 12 (1992).
[3] V. Arya, N. Garg, R. Khandekar, V. Pandit, Local search heuristics for k-median and facility location problems, in: Proceedings of the 33rd ACM Symposium on the Theory of Computing (STOC 2001), 2001.
[4] F. Barahona, F. Chudak, Solving large scale uncapacitated location problems, IBM Research Report RC 21515, IBM Research Division, T.J. Watson Research Center, Yorktown Heights, New York, 1999.
[5] J.E. Beasley, Obtaining test problems via Internet, Journal of Global Optimization 8 (1996) 429–433.
[6] F.A. Chudak, Improved approximation algorithms for uncapacitated facility location, in: Proceedings of the 6th Conference on Integer Programming and Combinatorial Optimization, 1998, pp. 180–194.
[7] C. Codognet, D. Diaz, Yet another local search method for constraint solving, in: AAAI Fall Symposium on Using Uncertainty within Computation, Cape Cod, MA, 2001.

[8] A.R. Conn, G. Cornuéjols, A projection method for the uncapacitated facility location problem, Mathematical Programming 46 (1990) 273–298.
[9] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, MIT Press, Cambridge, MA, 1990.
[10] G. Cornuéjols, G.H. Nemhauser, L. Wolsey, The uncapacitated facility location problem, in: Discrete Location Theory, Lecture Note in Artificial Intelligence (LNAI 1865), Wiley, 1990, pp. 119–171.
[11] D. Erlenkotter, A dual-based procedure for uncapacitated facility location: General solution procedures and computational experience, Operations Research 26 (1978) 992–1009.
[12] V. Filipovic, J. Kratica, D. Tosic, I. Ljubic, Fine grained tournament selection for the simple plant location problem, in: Proceedings of the 5th Online World Conference on Soft Computing in Industrial Applications WSC5, 2000, pp. 152–158.
[13] L.L. Gao, E.P. Robinson, Uncapacitated facility location: General solution procedures and computational experience, European Journal of Operations Research 76 (1994) 410–427.
[14] M. Guignard, A Langrangean dual ascent algorithm for simple plant location problems, European Journal of Operations Research 35 (1988) 193–200.
[15] P. Hansen, N. Mladenovic, An introduction to variable neighborhood search, in: S. Voss, S. Martello, I.H. Osman, C. Roucairol (Eds.), Meta-heuristics, Advances and Trends in Local Search Paradigms for Optimization, Kluwer Academic Publishers, 1998, pp. 433–458.
[16] K. Holmberg, Experiments with primal-dual decomposition and subgradient method for the uncapacitated facility location problem, Research Report LITH-MATH/OPT-WT-1995-08, Department of Mathematics, Linkoping Institute of Technology, Sweden, 1995.
[17] M. Koerkel, On the exact solution of large-scale simple plant location problems, European Journal of Operations Research 39 (1989) 157–173.
[18] J. Kratica, Improving performances of the genetic algorithm by caching, Computers and Artificial Intelligence 18 (3) (1999) 271–283.
[19] J. Kratica, V. Filipovic, V. Sesum, D. Tosic, Solving the uncapacitated warehouse location problem using a simple genetic algorithm, in: Proceedings of the XIV International Conference on Material Handling and Warehousing, 1996, pp. 3.33–3.37.
[20] J. Kratica, D. Tosic, V. Filipovic, Solving the uncapacitated warehouse location problem by sga with add-heuristic, in: XV ECPD International Conference on Material Handling and Warehousing, 1998.
[21] J. Kratica, D. Tosic, V. Filipovic, I. Ljubic, Solving the simple plant location problems by genetic algorithm, RAIRO Operations Research 35 (2001) 127–142.
[22] P. Mirchandani, D. Francis, Discrete Location Theory, John Wiley and Sons, New York, 1990.
[23] G. Ramalingam, Bounded incremental computation, PhD thesis, University of Wisconsin-Madison, 1993.
[24] D.B. Schmoys, E. Tardos, K. Aardal, Approximation algorithms for facility location problems, in: Proceedings of the 29th ACM Symposium on Theory of Computing, 1997, pp. 265–274.