Recursive Variable Neighborhood Search

Mohammad R. Raeesi N. and Ziad Kobti

***Abstract*—Variable Neighborhood Search (VNS) is one of the most recent metaheuristics to solve optimization problems. A new variant of VNS is introduced in this article called Recursive VNS (R-VNS). The proposed R-VNS incorporates recursive methods in order to improve both the exploration and exploitation capability of the basic VNS. The experiments show that the proposed R-VNS outperforms the basic VNS by offering better solutions as well as higher convergence rate. The case study considers classical Job Shop Scheduling Problem in order to evaluate both proposed methods.**

***Index Terms*—Job shop scheduling problem, recursive programing, variable neighborhood search.**

I. Introduction

The class of optimization problems is the set of problems where the goal is to make a system as effective as possible by optimizing its input variables. Optimization problems are divided into two categories, namely continuous optimization problems and combinatorial optimization problems. The focus of this paper is on the latter where input variables of the system are discrete. In combinatorial optimization problems a set of values with respect to all variables is called a solution, and the solution space is the set of all feasible solutions. The solution space is usually extremely large, but finite, and the goal is to find a solution with the optimal objective value. The objective could be either minimizing a cost function or maximizing a fitness function. In some large test problems where finding the optimal solution is difficult, near optimal solutions are sufficient.

There are different types of algorithms proposed to solve optimization problems. Heuristics are a class of approaches working on a solution space to find an optimal solution by incorporating a number of problem specific rules. Metaheuristics are general procedures using an iterative process to guide the operations of heuristics to deal with optimization problems. In other words, the goal of a metaheuristic is to build an efficient heuristic with a good performance on one problem domain. Various metaheuristics with different capabilities are introduced in literature to deal with both combinatorial and continuous optimization problems. One of the most recently introduced metaheuristics is Variable Neighborhood Search (VNS) proposed by Mladenovic and Hansen [1].

The procedure of VNS is to change the neighborhood systematically within a local search. Incorporating a number of neighborhood structures enables VNS to switch among them at the time of local search execution. When the local search finds a local optimal solution with respect to one neighborhood structure, VNS switches to another one to escape from that local optimum. This routine decreases the chance of trapping into local optimal solutions dramatically.

VNS is a general metaheuristic applicable in various areas such that it has been successfully applied in different combinatorial optimization problems such as the Traveling Salesman Problem [2], the Open Vehicle Routing problem [3], the *p*-Median problem [4], and the Graph problems [5].

The main contribution of this article is to introduce a new version of VNS which improves the searching capability of the canonical VNS. Since the new version uses recursive programming, it is called *Recursive VNS (R-VNS)* which is described later in detail. In order to evaluate the proposed R-VNS, Job Shop Scheduling Problem (JSSP) is considered as our case study, which is a combinatorial optimization problem.

The remainder of this article is organized as follows. Section II and Section III concisely describe VNS and the JSSP problem domain, respectively. Section IV represents the proposed R-VNS in detail, followed by illustrating the experiments designed to evaluate the proposed method, the discussion on the results and their comparison with the state-of-the-art methods in Section V. Finally, conclusion remarks are represented in Section VI.

II. Variable Neighborhood Search

Heuristics can be differentiated based on their strategy to find a good solution, which is either constructive or explorative strategy. Constructive heuristics incorporate a number of problem domain rules to construct the optimal or near optimal solutions, while explorative heuristics explore the solution space to find the optimal solution. Explorative heuristics suffer from their immature convergence. Since they only investigate the neighborhood of a solution, they can be trapped into the neighborhood region of a local optimal solution. They look around the neighborhood area to find better solution to move there, but since there is no better solution in the area, they never escape from there. Consequently they will converge into a local optimal solution instead of the global one. This issue is more challenging in large size problems where the chance of trapping into local optimal regions is high.

In order to overcome this limitation, explorative metaheuristics are proposed to design stronger heuristics. Metaheuristics may use one single solution or a population of solutions to do so. VNS is a single solution metaheuristic introduced to decrease the trapping chance of the existing heuristics. Although VNS is one of the most recently

Manuscript received September 20, 2013; revised December 26, 2013. This work is made possible by a grant from the National Science Foundation and NSERC Discovery No. 327482.

M. R. Raeesi N. and Z. Kobti are with School of Computer Science, University of Windsor, Windsor, ON N9B 3P4 (e-mail: raeesim@uwindsor.ca, kobti@uwindsor.ca).

introduced metaheuristics, a wide range of its successful application has been reported.

The basic routine of a VNS is to switch to another neighborhood structure when a local optimum with respect to one neighborhood structure is reached. The obtained local optimum is not necessarily an optimum with respect to the new neighborhood structure. Therefore, VNS can jump out of the local optimum region. VNS incorporates multiple neighborhood structures which are required to be complementary to each other. Otherwise, VNS will not be able to escape from local optimal regions. The complementary attribute of the neighborhood structures is a key factor for efficiency and effectiveness of a VNS.

The idea of VNS is inspired from the following facts:

- A local optimal solution with respect to one neighborhood structure could be a non-optimal solution with respect to another one.
- The global optimal solution is a local optimal solution with respect to all possible neighborhood structures.
- In general, local optimal solutions are often close to each other.

The last fact is obtained by empirical observations and consequently it cannot necessarily be relied upon for every test problem. However, it does imply that the local optimal solutions may have some useful information about the global optimal solution.

While the routine of VNS is fixed, there are different strategies to implement it. A survey on VNS methods and its applications is provided by Hansen *et al.* [6]. The authors described different implementations of VNS such as basic, reduced, general and skewed VNS. Almost all implementations include two subroutines, namely *Shake* and *LocalSearch*, such that the execution of a *Shake* function is followed by executing a *LocalSearch* subroutine, totally called a *Run*.

The pseudo-code for the basic version of VNS is represented in Fig. 1. The basic VNS starts with a random solution s (line 02), and with respect to the first neighborhood structure (line 04) it applies the *Shake* function to obtain another solution s' within the neighborhood of s (line 06). The *LocalSearch* subroutine is then applied on the neighbor solution s' to find out the local optimal solution s'' within its neighborhood with respect to the same neighborhood structure (line 07). The last step of each iteration of VNS is its selection mechanism (lines 08 through 13), in which if s'' is better than s, it continues with s'' starting from the first neighborhood, otherwise it switches to the next neighborhood structure. This routine continues until no improvement can be achieved with respect to all neighborhood structures, after which it restarts again with the best solution so far. Finally, this procedure ends as soon as the termination criteria (e.g. maximum CPU time, predefined number of iterations) are met.

It should be noted that the *Shake* subroutine randomly selects a solution s' within the neighborhood of solution s with respect to neighborhood structure N_k. While there are two strategies for the *LocalSearch* function, which are *First Improvement* and *Best Improvment*. In the former strategy, the *LocalSearch* method returns a solution as soon as it finds a better solution compared to the given solution, while in the latter, it searches the whole neighborhood area of the given solution and returns the best solution of the area. Although the latter approach may find better solution, it is extremely time-consuming.

```
PROCEDURE: Basic VNS
INPUT: Algorithm Parameters and Problem Specification
OUTPUT: Optimal or Near-Optimal Solutions
01  BEGIN
02    Generate an initial solution $s$
03    REPEAT
04      $k \leftarrow 1$
05      REPEAT
06        $s' \leftarrow Shake(s,k)$
07        $s'' \leftarrow LocalSearch(s',k)$
08        IF $f(s'') \le f(s)$
09          $s \leftarrow s''$
10          $k \leftarrow 1$
11        ELSE
12          $k \leftarrow k+1$
13        END IF
14      UNTIL ( $k > k_{max}$ )
15    UNTIL (termination criteria are met)
16    Output Solution $s$
17  END
```

Fig. 1. The pseudo-code of *Basic VNS*.

Overall, VNS has a very effective exploitation mechanism due to the definition of its procedure, but it suffers from an inefficient solution space exploration approach. Although it is able to find local optimal solutions in promising regions, it cannot explore the solution space effectively to find said regions.

III. Job Shop Scheduling Problem

The Job Shop Scheduling Problem (JSSP) is a well-known class of combinatorial optimization problems which are applicable in various research areas. In general, JSSP is the task of sequencing a number of jobs to be processed on a number of machines in order to optimize an evaluation function. The popular optimization function of JSSPs is *makespan* minimization which is to minimize the maximum completion time of all the jobs. As an open problem, JSSP is a good candidate to be used for the evaluation of optimization methods. Furthermore, Garey *et al.* [7] proved that JSSPs with more than two machines are NP-complete which implies that there is no exact algorithm to be able to find the optimal solution for all the scheduling problems in an acceptable time.

Due to different specification and various constraints, various versions of JSSP are introduced in literature. The general version is called classical JSSP as defined by Baker [8]. In classical JSSP, each problem is defined by a set of N jobs and M machines determining the problem size as $N \times M$. Jobs and Machines are denoted by J_i and m_k where i and k are the job index and machine index, respectively. Each job includes M operations with a fixed sequence to be processed on different machines such that each machine handles only one operation of a job. O_{ij} denotes the j^{th} operation of the i^{th} job.

The rules of classical JSSP can be summarized as follows:

- The operations of a job have to be processed in their predefined order, while the operations of different jobs are independent of each other.
- Each operation has to be evaluated on only one machine for a known processing time such that it cannot be interrupted.
- All jobs are available at the beginning without any due date.
- The set up time of machines and movement time of each job between two machines are considered negligible.

TABLE I: A SAMPLE 3×3 CLASSICAL JSSP

	Operation Index Jobs		
	O_1	O_2	O_3
J_1	m_2,1	m_1,2	m_3,3
J_2	m_1,2	m_3,2	m_2,2
J_3	m_2,2	m_3,4	m_1,1

Table I represents a sample classical JSSP with size 3×3. It illustrates the applicable machine and the corresponding processing time of each operation. The second operation of the first job, for instance, has to be completed on the first machine for 2 time units. A sample schedule for this test problem is illustrated in a Gantt chart in Fig. 2.

One of the important concepts in JSSP is critical operations which are defined as the operations such that any delay in their processing time increases the makespan of the schedule. Critical operations are determined on a critical path which is the longest path of consecutive operations starting from time zero to the makespan. A schedule may have more than one critical path and an operation may be located on different critical paths at the same time. A set of adjacent critical operations on the same machine is called a critical block. The first and the end operations of a block are called the block head and the block rear, respectively, and the operations in the middle of a block are called internal operations.

For instance, the critical path, the critical blocks and the critical operations of the sample schedule presented in Fig. 2 are as follows.

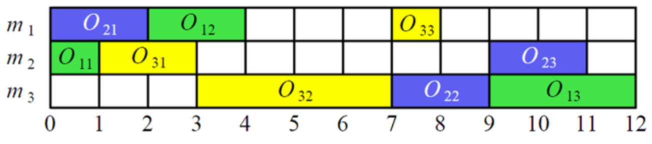

Fig. 2. A sample schedule for the sample problem represented in Table I.

$$\textit{Critical Path}: \quad O_{11} \prec O_{31} \prec O_{32} \prec O_{22} \prec O_{13}$$

$$\textit{Critical Blocks}: \quad \begin{cases} m_2 : O_{11} \prec O_{31} \\ m_3 : O_{32} \prec O_{22} \prec O_{13} \end{cases}$$

$$\textit{Critical Operations}: \quad \begin{cases} J_1 : O_{11}, O_{12} \\ J_2 : O_{22} \\ J_3 : O_{31}, O_{32} \end{cases}$$

As a result of these definitions, it is clear that the only way to decrease the makespan is to break up a critical path. Therefore the concepts of critical operations and critical blocks are very important to define efficient neighborhood structures. There are also a number of definitions in JSSP required to be clarified in order to be used later in neighborhood determination which include:

- Job's Operation Sequence: This sequence is the fixed operation sequence predefined for each job.
- Job-Successor Operation: This is the immediate next operation of the current operation on a job's operation sequence.
- Job-Predecessor Operation: The operation that is immediately before the current operation on a job's operation sequence.
- Machine's Operation Sequence: This is the sequence of operations that have to be processed on the same machine.
- Machine-Successor Operation: The operation exactly after the current operation on a machine's operation sequence.
- Machine-Predecessor Operation: This is the operation exactly before the current operation on a machine's operation sequence.

Job-successor operation of operation O is denoted by $JS(O)$, and its job-predecessor, its machine-successor and its machine-predecessor are denoted by $JP(O)$, $MS(O)$ and $MP(O)$, respectively. As the operation sequences of all jobs are predefined, the following equations are always correct provided $O_{i,j+1}$ and $O_{i,j-1}$ exist.

$$JS(O_{i,j}) = O_{i,j+1}$$
$$JP(O_{i,j}) = O_{i,j-1}$$

Another concept in JSSP which is very useful to limit the solution space is active schedule. Defined by Croce *et al.* [9], the schedules where there is no operation that can be started earlier without delaying another operation are active. A primary aspect of this concept is that the optimal solution is more likely an active solution, and if it is not, it has an equivalent active schedule which is optimal as well. Therefore - instead of exploring the whole solution space - it is more efficient to just search among active schedules. This concept is incorporated by various researchers with different mechanisms such as a gap reduction rule [10] and a priori knowledge [11].

IV. PROPOSED RECURSIVE VNS

Herein, a new version of VNS is proposed to improve upon the basic algorithm. The modified version called Recursive VNS (R-VNS) is the main contribution of this article which is described in Subsection A. Subsections B and C describe the solution representation and neighborhood structures for JSSP which are incorporated in our proposed method. The proposed method uses a more powerful fitness evaluation function which is represented in Subsection D.

A. Main Contribution

By the knowledge of the authors, all variants of VNS incorporate the *Shake* function followed by the *LocalSearch* subroutine with respect to one neighborhood structure (One *Run* with respect to each neighborhood structure). It means that the *LocalSearch* subroutine of two different neighborhood structures are not executed immediately after each other. The key contribution here is to define one *Run* with respect to all neighborhood structures which includes:

- A *RecursiveShake* function with respect to all neighborhood structures for a number of iterations.
- A *RecursiveLocalSearch* subroutine with respect to all neighborhood structures.

The *RecursiveShake* function calls the *Shake* function for all neighborhood structures consecutively such that the *Shake* subroutine with respect to the latter neighborhood structure is applied on the result of its application with respect to the former one. The pseudo-code of the *RecursiveShake* function is represented in Fig. 4 in which *ShakeIterations* denotes the number of iterations the *Shake* subroutine is applied with respect to each neighborhood structure.

```
PROCEDURE: RecursiveLocalSearch
INPUT: Current Solution $s'$
OUTPUT: A Local Optimal Solution
01  BEGIN
02    $k \leftarrow 1$
03    REPEAT
04        $s'' \leftarrow LocalSearch(s', k)$
05        IF $f(s'') \le f(s')$
06            $s' \leftarrow s''$
07            Output $RecursiveLocalSearch(s')$
08        ELSE
09            $k \leftarrow k+1$
10        END IF
11    UNTIL ( $k > k_{max}$ )
12    Output Solution $s'$
13  END
```

Fig. 3. The pseudo-code of *RecursiveLocalSearch*.

```
PROCEDURE: RecursiveShake
INPUT: Current Solution $s$ and Neighborhood $k$
OUTPUT: A Random Neighbor Solution
01  BEGIN
02    FOR ( ShakeIterations )
03        $s' \leftarrow Shake(s, k)$
04        $s \leftarrow s'$
05    END FOR
06    $k \leftarrow k+1$
07    IF $k \le k_{max}$
08        Output $RecursiveShake(s', k)$
09    END IF
10    Output Solution $s'$
11  END
```

Fig. 4. The pseudo-code of *RecursiveShake*.

Like *RecursiveShake*, the *RecursiveLocalSearch* function executes the *LocalSearch* subroutine with respect to all neighborhood structures consecutively, and if a better solution is reached, the *RecursiveLocalSearch* will be called recursively. Fig. 3 illustrates the pseudo-code of the *RecursiveLocalSearch*. Due to the recursive characteristic of both new functions, the new VNS version is called Recursive VNS, the pseudo-code of which is depicted in Fig. 5. R-VNS starts with a random solution, and applies the *RecursiveShake* function (line 05) followed by the *RecursiveLocalSearch* subroutine (line 06) on the current solution. Finally, a selection mechanism decides whether to keep the new solution or not (lines 07 through 09).

The main goal of incorporating recursive methods is to develop a more explorative and exploitative method. By incorporating recursive strategy, the proposed *RecursiveShake* procedure is able to explore further regions compared to the simple *Shake* method. This mechanism helps the search method to avoid from trapping into local optimal regions. The *RecursiveLocalSearch* method is designed to improve the exploitation mechanism of the basic VNS. This method exploits a region more deeply until it cannot find a better solution with respect to all neighborhood structures without disturbing the current solution.

```
PROCEDURE: RecursiveVNS
INPUT: Algorithm Parameters and Problem Specification
OUTPUT: Optimal or Near-Optimal Solutions
01  BEGIN
02    Generate an initial solution $s$
03    REPEAT
04        $k \leftarrow 1$
05        $s' \leftarrow RecursiveShake(s, k)$
06        $s'' \leftarrow RecursiveLocalSearch(s')$
07        IF $f(s'') \le f(s)$
08            $s \leftarrow s''$
09        END IF
10    UNTIL ( termination criteria are met )
11    Output Solution $s$
12  END
```

Fig. 5. The pseudo-code of *RecursiveVNS*.

Overall the proposed *RecursiveShake* and *RecursiveLocalSearch* methods improve the exploration and exploitation capabilities of the basic VNS, respectively. Therefore, it is expected that the proposed R-VNS incorporating these methods offers better solutions as well as better convergence rates compared to the basic VNS.

In order to incorporate R-VNS, a solution representation and a number of neighborhood structures should be determined. The selected representation and neighborhood structures for experiments are represented as follows.

B. Solution Representation

In literature, there are various representations for JSSP with their own advantages and disadvantages. One of the well-known representation is introduced by Bierwirth [12] which is mathematically called permutation with repetition. Permutation with repetition representation is an operation-based representation encoding a schedule into one string of job indices. The index of each job is repeated p times where p is the number of its operations. The total length of this string for a test problem is equal to the total number of operations in that problem. Each job index denotes one operation of the corresponding job and the operations within the same job are distinguished with the occurrence of the same index. The following example represents more details about this representation. Consider

$$\{1,2,1,3,3,2,2,1,3\}$$

as a sample permutation encoded by permutation with repetition representation for the sample problem illustrated in Table I. This permutation is decoded into the following operation sequence.

$$O_{11} \prec O_{21} \prec O_{12} \prec O_{31} \prec O_{32} \prec O_{22} \prec O_{23} \prec O_{13} \prec O_{33}$$

This operation sequence produces the following schedule depicted in Fig. 2.

$$m_1 : O_{21} \prec O_{12} \prec O_{33}$$
$$m_2 : O_{11} \prec O_{31} \prec O_{23}$$
$$m_3 : O_{32} \prec O_{22} \prec O_{13}$$

The advantage of this representation is that all the possible permutations are feasible solutions, and no repair mechanism is required. But this representation suffers from its inefficient n to 1 mapping such that there could be a large number of different permutations with the same encoded schedule. Compared to Machine Operation List (MOL) representation [13] as an example, the number of all possible permutations in MOL representation is much smaller than that of the permutation with repetition representation, while both representations encode the same solution space. Furthermore, the MOL's possible permutations include some infeasible solutions in addition to all feasible solutions included in the solution space. This comparison shows how inefficient the mapping of the permutation with repetition representation is. However, due to its very efficient decoding and encoding procedures, it has been incorporated by various researchers.

C. Neighborhood Structures

There are various neighborhood structures proposed for JSSP. Blazewicz *et al.* [14] provided a review on the techniques proposed to deal with JSSPs. In this review, six popular neighborhood structures are presented which are called $N1$ to $N6$ by the authors. A brief description of these neighborhoods is provided as follows:

- Neighborhood Structure $N1$: This structure introduced by Van Laarhoven *et al.* [15] is a very simple structure such that swapping two adjacent critical operations is considered as a valid move to generate new schedule. The neighborhood area for this structure is pretty large.
- Neighborhood Structure $N2$: This neighborhood structure is defined by Matsuo *et al.* [16] in which swapping two critical operations p and q is considered as a valid move if either p is a block head or q is a block rear. In addition, two additional moves are considered to provide more chance to obtain a schedule with lower makespan including swapping $MP(JP(p))$ and $JP(p)$, and swapping $JS(q)$ and $MS(JS(q))$.
- Neighborhood Structure $N3$: This structure is incorporated by Dell'Amico and Trubian [17] in which all permutations of three operations $\{MP(p), p, q\}$ and $\{p, q, MS(q)\}$ have been considered such that p and q are adjacent critical operations swapped in the provided permutations. The neighborhood area of this structure is finite but very large. Neighborhood structure $N3'$ is a limited version of $N3$ in which either p or q should be a block end.
- Neighborhood Structure $N4$: Represented by Dell'Amico and Trubian [17], $N4$ considers moving an internal operation to the very beginning or to the very end of a block.
- Neighborhood Structure $N5$: Nowicki and Smutnicki [18] introduced the smallest neighborhood area using neighborhood structure $N5$ in which only the first two operations or the last two operations of a critical block are the candidates for swap operation.
- Neighborhood Structure $N6$: The extension of all previously described neighborhood structures is proposed by Balas and Vazacopoulos [19]. Considering p and q as two critical operations on a critical block, a valid move in $N6$ is defined as moving q right before p if $JP(p)$ belongs to the critical path, and moving p right after q if $JS(q)$ belongs to the critical path. The authors called these moves *backward interchange* and *forward interchange*, respectively.

In the proposed R-VNS, neighborhood structures $N4$, $N5$ and $N6$ are incorporated with different strategies for *RecursiveShake* and *RecursiveLocalSearch* procedures which are presented in Equations (1) and (2), respectively. Furthermore, the neighborhood structure $N5$ is considered as a nested neighborhood structure for *RecursiveLocalSearch* when it cannot find a better solution.

RecursiveShake:

$$k_{max} = 3 \qquad N_k = \begin{cases} N4, & \text{if } k = 1 \\ N5, & \text{if } k = 2 \\ N6, & \text{if } k = 3 \end{cases} \tag{1}$$

RecursiveLocalSearch:

$$k_{max} = 2 \qquad N_k = \begin{cases} N4, & \text{if } k = 1 \\ N6, & \text{if } k = 2 \end{cases} \tag{2}$$

It should be mentioned here that there are not any pre-processing or post-processing procedures incorporated for any move in each neighborhood structure. The move is just moving one index either forward or backward in the solution representation string.

D. Evaluation Function

In order to evaluate each solution in an optimization problem, the simple way is to find out only the objective value of that solution. This approach works well for evaluation but not for comparison. When comparing a number of solutions with the same objective value, according to the simple approach one of them has to be selected randomly, while a better way is to consider other characteristics of those solutions.

For the solution evaluation in the proposed method, a Priority-Based Fitness Function (PBFF) [20] is incorporated. Since the proposed method is going to be applied on classical JSSPs, only two priorities are defined for the PBFF. The first priority is the makespan, and the second one is the number of critical machines such that both priorities have to be minimized. Using this evaluation function, in case of a tie in comparison, the solution which has the lowest number of critical machines is selected.

V. Experiments and Discussion

The proposed method is implemented using the *Java* programming language (version 1.6.0.18) and experiments are done on a system with *Intel(R) Xeon(R) 2.27GHz CPU (16 Core)* and *24GB RAM*. As mentioned before (in order to evaluate the method), they are applied on classical JSSP. As one of the well-known classical JSSP benchmarks, the case study considers the data set introduced by Lawrence [21]. This benchmark (denoted by *LA*) includes 40 test problems with different size and complexity levels. While almost all the state-of-the-art methods, as well as the proposed method, are able to find the optimal solution for 28 test problems in every run, the remaining 12 problems are still considered challenging. In the experiments the algorithm is evaluated based on 50 independent runs for each test problem.

Table II represents the algorithm parameters adjusted by incorporating extensive experiments. The proposed R-VNS runs for 20,000 iterations incorporating neighborhood structures $N4$, $N5$, and $N6$. To have a better representation of the table, neighborhood structure and nested neighborhood structure are denoted by NS and NNS, respectively.

Regarding to the number of shake iterations in the proposed *RecursiveShake* procedure two strategies are considered, namely Fixed *ShakeIterations* (*FSI*) and Variable *ShakeIterations* (*VSI*). In the former strategy the parameter *ShakeIterations* is set to 2 and remains the same for all iterations of one experiment, while in the latter it is set to a number between 2 and 5 which is selected randomly for each iteration.

The proposed R-VNS in two versions with different *ShakeIterations* strategies is applied on the challenging *LA* test problems, the results of which are represented in Table III. Both versions show almost the same performance such that their best found solutions are similar for all the 12 test problems, except problem *LA38* where the VSI version finds the optimal solution. In addition to the optimal solution for problem *LA38*, the VSI version offers better average, median and worst solutions for almost all 12 test problems. Therefore, it is possible to say that the R-VNS (VSI) slightly outperforms R-VNS (FSI).

In order to show the performance of the proposed R-VNS, the basic VNS is also applied on the same problems with the same configuration to have a fair comparison. As represented in Table III both versions of R-VNS outperform the basic VNS by finding better solutions. The basic VNS can only find the optimal solution for three test problems out of 12, while the proposed R-VNS (FSI) and R-VNS (VSI) offer the optimal solutions for 7 and 8 test problems, respectively. Furthermore, the statistical analysis of the results shows much better average, median and worse solutions for the proposed R-VNS compared to the basic VNS.

TABLE II: Asjusted Parameters of the Proposed R-VNS

Proposed R-VNS	
Parameter	**Value**
MaxIteration	20,000
$Shake_{NS}$	$N4$, $N5$, and $N6$
$LocalSearch_{NS}$	$N4$ and $N6$
$LocalSearch_{NNS}$	$N5$
FSI	2
VSI	$\{2,3,4,5\}$

NS: Neighborhood Structure
NNS: Nested Neighborhood Structure

Overall the results show that the proposed R-VNS outperform the basic VNS by offering better solutions as well as improving the convergence rate. Therefore, the results confirm that the proposed *RecursiveShake* and *RecursiveLocalSearch* methods are capable to improve the exploration and exploitation mechanisms of the basic VNS, respectively.

In order to demonstrate the performance of the proposed methods, the state-of-the-art methods in the area are considered for comparison including three recently published methods. The first method which is proposed by Zobolas *et al.* [22] is a hybridization of a Genetic Algorithm (GA) and a VNS which incorporates a Differential Evolution to generate an initial population. The authors called it as a hybrid Evolutionary Algorithm (hEA). The two other methods are our recently published VNS and Memetic Algorithm (MA) [23]. The published MA is a GA joined with a VNS.

In order to evaluate the proposed over all the test problems, the *Error Rate (ER)* parameter is incorporated, which is the percentage error from the optimal solution. ER is calculated using Equation (3) where C denotes the best solution found by the algorithm and BK denotes the best-known solution.

$$ER = \frac{C - BK}{BK} \times 100\% \tag{3}$$

The results of all three methods on the most challenging LA problems are illustrated in Table IV. The ER parameter is also incorporated here in order to have a fair comparison over different test problems. The values within brackets and the values illustrated on the last row of the table represent the ER values and their averages over 12 test problems, respectively. The ER values demonstrate that both proposed R-VNS methods offers competitive solutions compared to the state-of-the-art methods. Overall the proposed R-VNS (VSI) is an effective method such that it offers a very low average ER for the most challenging LA problems (as low as 0.20%).

TABLE III: RESULTS ON THE CHALLENGING PROBLEMS OF LA BENCHMARK

Problem	Method	Best	Average	SD	Median	Worst
LA20	Basic VNS	907	907.00	0.00	907.0	907
10×10	R-VNS (FSI)	**902**	906.60	1.37	907.0	907
902	R-VNS (VSI)	**902**	906.60	1.37	907.0	907
LA21	Basic VNS	**1046**	1065.66	10.50	1067.0	1093
15×10	R-VNS (FSI)	**1046**	1061.00	9.28	1058.0	1084
1046	R-VNS (VSI)	**1046**	1057.00	7.48	1056.0	1077
LA24	Basic VNS	938	947.56	6.06	946.0	970
15×10	R-VNS (FSI)	**935**	945.02	5.07	946.0	961
935	R-VNS (VSI)	**935**	944.46	5.08	946.0	957
LA25	Basic VNS	**977**	986.62	6.36	984.0	1006
15×10	R-VNS (FSI)	**977**	984.84	5.10	984.0	1004
977	R-VNS (VSI)	**977**	983.48	3.92	983.5	993
LA27	Basic VNS	1237	1253.48	9.38	1253.0	1269
20×10	R-VNS (FSI)	**1235**	1251.20	10.63	1248.5	1269
1235	R-VNS (VSI)	**1235**	1249.86	10.25	1247.5	1269
LA28	Basic VNS	**1216**	1219.40	5.81	1216.0	1234
20×10	R-VNS (FSI)	**1216**	1217.78	4.35	1216.0	1234
1216	R-VNS (VSI)	**1216**	1217.44	2.92	1216.0	1227
LA29	Basic VNS	1173	1195.58	12.94	1194.5	1232
20×10	R-VNS (FSI)	1163	1186.80	15.65	1188.0	1228
1152	R-VNS (VSI)	1163	1189.34	12.56	1189.5	1221
LA36	Basic VNS	1281	1294.94	7.47	1292.0	1315
15×15	R-VNS (FSI)	1274	1291.14	7.06	1291.0	1308
1268	R-VNS (VSI)	1274	1290.50	6.35	1291.0	1299
LA37	Basic VNS	1400	1424.28	14.32	1424.0	1457
15×15	R-VNS (FSI)	**1397**	1419.28	14.09	1418.0	1455
1397	R-VNS (VSI)	**1397**	1416.16	9.68	1418.0	1433
LA38	Basic VNS	1202	1237.04	14.66	1237.5	1263
15×15	R-VNS (FSI)	1201	1232.44	15.66	1232.0	1260
1196	R-VNS (VSI)	**1196**	1230.54	14.29	1231.5	1259
LA39	Basic VNS	1240	1248.36	5.55	1249.0	1268
15×15	R-VNS (FSI)	1239	1246.84	5.29	1248.0	1259
1233	R-VNS (VSI)	1239	1246.92	5.05	1248.0	1258
LA40	Basic VNS	1228	1241.24	5.38	1242.0	1254
15×15	R-VNS (FSI)	1228	1241.04	6.54	1241.5	1263
1222	R-VNS (VSI)	1228	1240.40	6.75	1240.5	1252

TABLE IV: COMPARISON WITH THE STATE-OF-THE-ART METHODS

Problem	BK	hEA [22]	VNS [23]	MA [23]	R-VNS (FSI)	R-VNS (VSI)
LA20	**902**	-	**902 (0.00%)**	**902 (0.00%)**	**902 (0.00%)**	**902 (0.00%)**
LA21	**1046**	**1046 (0.00%)**	**1046 (0.00%)**	**1046 (0.00%)**	**1046 (0.00%)**	**1046 (0.00%)**
LA24	**935**	**935 (0.00%)**	**935 (0.00%)**	**935 (0.00%)**	**935 (0.00%)**	**935 (0.00%)**
LA25	**977**	**977 (0.00%)**	979 (0.20%)	**977 (0.00%)**	**977 (0.00%)**	**977 (0.00%)**
LA27	**1235**	1236 (0.08%)	1244 (0.73%)	1238 (0.24%)	**1235 (0.00%)**	**1235 (0.00%)**
LA28	**1216**	1224 (0.66%)	**1216 (0.00%)**	**1216 (0.00%)**	**1216 (0.00%)**	**1216 (0.00%)**
LA29	**1152**	1160* (0.69%)	1169 (1.48%)	1163 (0.95%)	1163 (0.95%)	1163 (0.95%)
LA36	**1268**	**1268 (0.00%)**	1291 (1.81%)	1281 (1.03%)	1274 (0.47%)	1274 (0.47%)
LA37	**1397**	1408 (0.79%)	**1397 (0.00%)**	**1397 (0.00%)**	**1397 (0.00%)**	**1397 (0.00%)**
LA38	**1196**	1202 (0.50%)	1208 (1.00%)	1208 (1.00%)	1201 (0.42%)	**1196 (0.00%)**
LA39	**1233**	**1233 (0.00%)**	1241 (0.65%)	1241 (0.65%)	1239 (0.49%)	1239 (0.49%)
LA40	**1222**	1229 (0.57%)	1233 (0.90%)	1233 (0.90%)	1228* (0.49%)	1228* (0.49%)
Average ER		0.30%	0.56%	0.32%	0.24%	0.20%

VI. CONCLUSIONS

The main contribution of this article is to introduce a new version of VNS called Recursive VNS (R-VNS). R-VNS calls the *Shake* and *LocalSearch* functions recursively. The key idea of the proposed R-VNS is to call the *Shake* subroutine with respect to different neighborhood structures immediately after each other. This mechanism forces the search method to explore more regions. Calling the *LocalSearch* method recursively, improves the exploitative mechanism of the basic VNS. Applying the proposed R-VNS on a number of classical JSSP shows that the proposed R-VNS outperforms the basic VNS by offering better solutions as well as improving the convergence rate.

However, there is still some possibility to improve the proposed method. One direction is to investigate the effect of *ShakeIterations* parameter over different iterations. A dynamic strategy (*DSI*) may outperform both FSI and VSI strategies. Another direction is to incorporate an EA to be joined with the proposed R-VNS in order to make it highly explorative.

REFERENCES

[1] N. Mladenovic and P. Hansen, "Variable neighborhood search," *Computers and Operations Research*, vol. 24, pp. 1097-1100, 1997.

[2] A. Felipe, M. T. Ortuno, and G. Tirado, "The double traveling salesman problem with multiple stacks: a variable neighborhood search approach," *Computers and Operations Research*, vol. 36, no. 11, pp. 2983-2993, 2009.

[3] K. Fleszar, I. H. Osman, and K. S. Hindi, "A variable neighbourhood search algorithm for the open vehicle routing problem," *European Journal of Operational Research*, vol. 195, pp. 803-809, 2009.

[4] K. Fleszar and K. S. Hindi, "An effective VNS for the capacitated p-median problem," *European Journal of Operational Research*, vol. 191, no. 3, pp. 612-622, 2008.

[5] J. Brimberg, N. Mladenovic, D. Urosevic, and E. Ngai, "Variable neighborhood search for the heaviest k-subgraph," *Computers and Operations Research*, vol. 36, no. 11, pp. 2885-2891, 2009.

[6] P. Hansen, N. Mladenovic, and J. A. M. Perez, "Variable neighbourhood search: methods and applications," *4OR: A Quarterly Journal of Operations Research*, vol. 6, no. 4, pp. 319-360, 2010.

[7] M. R. Garey, D. S. Johnson, and R. Sethi, "The complexity of flowshop and jobshop scheduling," *Mathematics of Operations Research*, vol. 1, pp. 117-129, 1976.

[8] K. R. Baker, *Introduction to Sequencing and Scheduling*, Wiley, 1974.

[9] F. D. Croce, R. Tadei, and G. Volta, "A genetic algorithm for the job shop problem," *Computers in Operations Research*, vol. 22, pp. 15-24, 1995.

[10] S. M. K. Hasan, R. Sarker, D. Essam, and D. Cornforth, "Memetic algorithms for solving job-shop scheduling problems," *Memetic Computing*, vol. 1, pp. 69-83, 2008.

[11] R. L. Becerra and C. A. Coello, "A cultural algorithm for solving the job-shop scheduling problem," in *Proc. Knowledge Incorporation in Evolutionary Computation, Studies in Fuzziness and Soft Computing*, 2005, vol. 167, pp. 37-55.

[12] C. Bierwirth, "A generalized permutation approach to job shop scheduling with genetic algorithms," *OR Spectrum - Special Issue on Applied Local Search*, vol. 17, no. 2-3, pp. 87-92, 1995.

[13] M. R. Raeesi N. and Z. Kobti, "A machine operation lists based memetic algorithm for job shop scheduling," in *Proc. IEEE Congress on Evolutionary Computation (CEC)*, New Orleans, LA, USA, 2011 pp. 2436-2443.

[14] J. Blazewicz, W. Domschke, and E. Pesch, "The job shop scheduling problem: Conventional and new solution techniques," *European Journal of Operational Research*, vol. 93, no. 1, pp. 1–33, 1996.

[15] P. J. M. Van Laarhoven, E. H. L. Aarts, and J. K. Lenstra, "Job shop scheduling by simulated annealing," *Operations Research*, vol. 40, no. 1, pp. 113-125, 1992.

[16] H. Matsuo, C. J. Suh, and R. S. Sullivan, "A controlled search simulated annealing method for the general job shop scheduling problem," Working paper 03-04-88, University of Texas at Austin, 1988.

[17] M. Dell'Amico and M. Trubian, "Applying tabu search to the job-shop scheduling problem," *Annals of Operations Research*, vol. 41, no. 3, pp. 231-252, 1993.

[18] E. Nowicki and C. Smutnicki, "A fast taboo search algorithm for the job shop scheduling problem," *Management Science*, vol. 42, no. 6, pp. 797-813, 1996.

[19] E. Balas and A. Vazacopoulos, "Guided local search with shifting bottleneck for job shop scheduling," *Management Science*, vol. 44, no. 2, pp. 262-275, 1998.

[20] M. R. Raeesi N. and Z. Kobti, "A memetic algorithm for job shop scheduling using a critical-path-based local search heuristic," *Memetic Computing*, vol. 4, no. 3, pp. 231-245, 2012.

[21] S. Lawrence, "Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques," Master's thesis, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1984.

[22] G. I. Zobolas, C. D. Tarantilis, and G. Ioannou, "A hybrid evolutionary algorithm for the job shop scheduling problem," *Journal of the Operational Research Society*, vol. 60, pp. 221-235, 2009.

[23] M. R. Raeesi N. and Z. Kobti, "Incorporating a genetic algorithm to improve the performance of variable neighborhood search," in *Proc. 4th World Congress on Nature and Biologically Inspired Computing (NaBIC)* , Mexico City, Mexico, 2012, pp. 144-149.

Mohammad R. Raeesi N. accomplished his bachelor of science degree in information technology at Sharif University of Technology, Tehran, Iran in 2008. Consequently in 2010, he completed his master of applied science degree in the field of 3D image processing at Computer and Electrical Engineering Department in University of Windsor, Windsor, ON, Canada. He pursued his scientific career by starting his Ph.D. program at School of Computer Science in University of Windsor. His main research interest is evolutionary computation such that he has explored different characteristics of evolutionary algorithms specially cultural algorithm.

Ziad Kobti completed his doctorate (Ph.D.) in computer science in 2004 from Wayne State University, Michigan, USA, in the field of artificial intelligence (AI). He completed his Bachelor of Science, Honours degree (B.Sc.H.) majoring in Biological and Computer Sciences in 1996 followed by a Master of Science degree (M.Sc.) in computer science both from the University of Windsor, Ontario, Canada. In January 2005 he was appointed to a tenure track assistant professor position at Windsor. He was also appointed an adjunct faculty at Wayne State shortly afterwards in order to pursue further research collaborations. In 2009 he was appointed as an adjunct researcher at Washington State University, Pullman, Washington with the department of Anthropology. Dr. Kobti received his early tenure in July 2009 and promoted to associate professor in July 2011. Dr. Kobti was then appointed as the director to the School of Computer Science to begin July 2012 for a 5 years term.