Adaptive Neighbourhood Search for the Component Deployment Problem

Aldeida Aleti[1] and Madalina Drugan[2]

[1] Faculty of Information Technology, Monash University, Australia
aldeida.aleti@monash.edu
[2] Artificial Intelligence Lab, Vrije Universiteit Brussel, Belgium
mdrugan@vub.ac.be

Abstract. Since the establishment of the area of search-based software engineering, a wide range of optimisation techniques have been applied to automate various stages of software design and development. Architecture optimisation is one of the aspects that has been automated with methods like genetic algorithms, local search, and ant colony optimisation. A key challenge with all of these approaches is to adequately set the ballance between exploration of the search space and exploitation of best candidate solutions. Different settings are required for different problem instances, and even different stages of the optimisation process. To address this issue, we investigate combinations of different search operators, which focus the search on either exploration or exploitation for an efficient variable neighbourhood search method. Three variants of the variable neighbourhood search method are investigated: the first variant has a deterministic schedule, the second variant uses fixed probabilities to select a search operator, and the third method adapts the search strategy based on feedback from the optimisation process. The adaptive strategy selects an operator based on its performance in the previous iterations. Intuitively, depending on the features of the fitness landscape, at different stages of the optimisation process different search strategies would be more suitable. Hence, the feedback from the optimisation process provides useful guidance in the choice of the best search operator, as evidenced by the experimental evaluation designed with problems of different sizes and levels of difficulty to evaluate the efficiency of varying the search strategy.

Keywords: Adaptive neighbourhood search, component deployment optimisation

1 Introduction

One of the main aims of search-based software engineering (SBSE) is the automation of software design and development [14]. Ideally, the system developer would only have to submit requirements models, which would be used to generate the entire software system. Although many stages of software design and development have been automated, such as architecture design optimisation [3,1], code generation and repair [25], and software test case generation [4], a system that performs the enormous task of completely automating the process of software development from requirements does not exist, at least not yet. Nevertheless, each individual effort in the automation of specific stages brings us one step closer to the ultimate goal of SBSE.

The decision regarding the architecture of the system, being one of the most creative and important steps of software development [3] affects the quality of the final software system. Designing a software architecture that does not only satisfy the functional requirements, but that is at the same time optimal in terms of quality attributes, such as performance and reliability is not an easy task. The concept of software architecture is defined as 'the fundamental concepts or properties of a system in its environment embodied in elements, relationships, and in the principles of its design and evolution' [15]. In this paper we focus on embedded systems, where the architecture is composed of software components, hardware units, interactions of software components, and communications between hardware units. The allocation of software components into the hardware units and the assignment of interactions to the communication network, known as the component deployment problem is among design decision that have to be made at this stage. The search space of this problem is very large. For instance, in a system with 10 hardware units and 60 software components there are $20^{60} \approx 1.15 \times 10^{78}$ possible options, which are clearly beyond a human's capacity to handle at a reasonable amount of time.

This has lead to the application of a wide range of search-based methods in software architecture design [3], to deal with the complexity of software systems, the enormous design space and the effect of design decisions on quality attributes. Furthermore, search-based methods may produce architectures that a system designer would have not been able to think of, helping with the creative process. These efforts include methods like linear programming [23,9], genetic algorithms [20,1,17], and local search [13]. The majority of these approaches consider experimental studies to determine the success of the optimisation strategy based on a set of selected problem instances. The no-free-lunch theorems tell us that 'for any algorithm, any elevated performance over one class of problems is exactly paid for in performance over another class' [26].

More specifically, the performance of an optimisation algorithm highly depends on the fitness landscape of the targeted optimisation problem. Examples of landscape features are the number and distribution as well as the sizes of the optima, the location of the global optimum, and plateaus. A fitness landscape with a single optimum is easy to search with a local search method. On the other extreme, problems with many local optima and an isolated global optimum create a fitness landscape that is rugged and hard to explore. A plateau symbolises the presence of neighbouring solutions with equal fitness, where the progress of a search algorithm potentially stagnates.

The choice of neighbourhood structure determines whether the fitness landscape is easy to search. For example, if the fitness difference between any two neighbouring solutions is on average small then the landscape is more likely to be suited for a wide range of local search operators. In contrast, if significant fitness difference is encountered in the neighbourhood, different operators will produce different quality results, and the choice of the operator becomes important.

For new problems, like the Component Deployment, the structure of the fitness landscape that arises from different search operators is not known. A poor choice of the neighbourhood operator may lead to suboptimal algorithm performance. Ideally, the neighbourhood operator should be adapted during the search-based on the structure of the fitness landscape. In this work, we investigate three strategies for varying the neighbourhood operator: a deterministic schedule, where the change is controlled by a deterministic rule, a variable schedule, where operators are selected based on predefined probabilities, and an adaptive strategy, which uses feedback from the optimisation process.

2 Component Deployment Optimisation

The component deployment problem refers to the allocation of software components to the hardware nodes, and the assignment of inter-component communications to network links. Formally, we define the software components as $\mathcal{C} = \{c_1, c_2, ..., c_n\}$, $n \in \mathbb{N}$. The execution of the software system is initiated in one software component (with a given probability), and during its execution uses many other components connected via communication links, which are assigned with a transition probability [16]. A software component has a memory size sz expressed in KB (kilobytes), workload wl, which is the computational requirement of a component expressed in MI (million instructions), and initiation probability q_0, which is the probability that the execution of a system starts from the component. Software components interact to perform various tasks. Each interaction from component c_i to c_j is annotated with the following properties: (i) data size ds_{ij} in kilobytes, referring to the amount of data transmitted from software component c_i to c_j during a single communication event, and (ii) next-step probability p_{ij} the probability that the execution of component c_i ends with a call to component c_j.

The hardware architecture is composed of a distributed set of hardware hosts, denoted as $\mathcal{H} = \{h_1, h_2, ..., h_m\}$, $m \in \mathbb{N}$. Each hardware host is annotated with the following properties: (i) memory capacity (cp) expressed in kilobytes, (ii) processing speed (ps), which is the instruction-processing capacity of the hardware unit, expressed in million instructions per second (MIPS), and (iii) failure rate (fr), which characterises the probability of a single hardware unit failure [7].

The hardware hosts are connected via links denoted as $\mathcal{N} = \{n_1, n_2, ...n_s\}$, with the following properties: (i) data rate (dr_{ij}), which is the data transmission rate of the bus, expressed in kilobytes per second (KBPS), and (ii) failure rate (fr_{ij}) is the exponential distribution characterising the data communication failure of each link.

The way the components are deployed affects many aspects of the final system, such as the processing speed of the software components, how much hardware is used or the reliability of the execution of different functionalities [21,5], which constitute the quality attributes of the system. Formally, the component deployment problem is defined as $D = \{d \mid d : \mathcal{C} \to \mathcal{H}\}$, where D is the set of all functions assigning components to hardware resources.

2.1 Objective Function

The reliability evaluation obtains the mean and variance of the number of visits of components in a single execution and combines them with the failure parameters of the components. Failure rates of *execution elements* can be obtained from the hardware parameters, and the time taken for the execution is defined as a function of the software-component workload and processing speed of its hardware host. The reliability of a component c_i can be computed by Equation 1, where $d(c_i)$ denotes the hardware host where component c_i is deployed.

$$R_i = e^{-\mathrm{fr}_{d(c_i)} \cdot \frac{\mathrm{wl}_i}{\mathrm{ps}_{d(c_i)}}}. \tag{1}$$

where $d(c_i)$ is the deployment function that returns the hardware host where component c_i has been deployed. The reliability of a *communication element* is characterised by the failure rates of the hardware buses and the time taken for communication, defined as a function of the bus data rates *dr* and data sizes *ds* required for software communication. The reliability of the communication between component c_i and c_j is defined as

$$R_{ij} = e^{-\mathrm{fr}_{d(c_i)d(c_j)} \cdot \frac{\mathrm{ds}_{ij}}{\mathrm{dr}_{d(c_i)d(c_j)}}}. \tag{2}$$

The probability that a software system produces the correct output depends on the number of times it is executed. The *expected number of visits* for each component $v : C \to \mathbb{R}_{\geq 0}$ is $v_i = q_i + \sum_{j \in \mathcal{I}} v_j \cdot p_{ji}$, where $\mathcal{I}$ denotes the index set of all components. The transfer probabilities p_{ji} can be written in a matrix form $P_{n \times n}$, where n is the number of components. Similarly, the execution initiation probabilities q_i can be expressed with matrix $Q_{n \times 1}$. The matrix of expected number of visits for all components $V_{n \times 1}$ can be calculated as $V = Q + P^T \cdot V$.

The reliability of a software system is also influenced by the failure rate of the network links used during the execution of the system. The more frequently the network is used, the higher is the probability of producing an incorrect output. It should be noted that the execution of a software system is never initiated in a network link, and the only predecessor of link l_{ij} is component c_i. Hence, the expected number of visits of network links $v : C \times C \to \mathbb{R}_{\geq 0}$ is calculated as $v_{ij} = v_i \cdot p_{ij}$. Finally, the reliability of a deployment architecture $d \in D$ is calculated as:

$$R = \prod_{i=1}^{n} R_i^{v_i} \prod_{i,j \text{ (if used)}} R_{ij}^{v_{ij}}. \tag{3}$$

2.2 Constraints

The problem is naturally constrained, since not all possible deployment architectures can be feasible. For the purpose of this work, we consider three constraints: allocation, memory and communication.

Allocation constraint takes care of the allocation of all software components into hardware resources. Formally, this constraint is modelled as

$$\sum_{j=1}^{m} x_{ij} = 1, \quad \forall i = 1, \ldots, n, \tag{4}$$

where x_{ij} is 1 if the software component i is deployed on the hardware unit j, and 0 otherwise.

Hardware memory capacity deals with the memory requirements of software components and makes sure that there is available memory in the hardware units. Processing units have limited memory, which enforces a constraint on the possible components that can be deployed into each hardware host. Formally, the memory constraint is defined as

$$\sum_{i=1}^{n} \mathrm{sz}_i x_{ij} \leq \mathrm{cp}_j, \quad \forall j \in \{1, \ldots, m\}. \tag{5}$$

Communication constraint is responsible for the communication between software components. If the transition probability between two software components i and j is positive, $p_{ij} > 0$, these two components will communicate with a certain probability. Therefore, either they should be deployed on the same hardware unit, or on different units that are connected with a communication link (bus) with a positive data rate, $\mathrm{dr}_{ij} > 0$. This is modelled as follows:

$$x_{ik} + x_{jl} \leq 1, \quad \text{if } p_{ij} > 0 \text{ and } \mathrm{dr}_{kl} \leq 0. \tag{6}$$

2.3 Related Work

For many decades, researchers have been developing evermore sophisticated algorithms to solve the component deployment problem [3]. Notable examples are genetic algorithms [19,18], ant colony optimisation [24,5] and heuristics [6]. Aleti et al. [5] formulated the component deployment problem as a biobjective optimisation problem with data transmission reliability and communication overhead as objectives. Memory capacity constraints, location and colocation constraints were considered in the formulation, which was solved using P-ACO [12] as well as MOGA [11]. P-ACO was found to produce better solutions in the initial optimisation stages, whereas MOGA continued to produce improved solutions long after P-ACO had stagnated.

A Bayesian learning method was developed by Aleti and Meedeniya [6] and applied to the formulation defined by Aleti et al. [5]. The probabilities of a solution being part of the non-dominated set was calculated as the ratio of non-dominated solutions produced in the current generation and the overall number of solutions in the generation. Compared to NSGA-II [10] and P-ACO, the Bayesian method was found to produce approximation sets with higher hypervolume values.

Meedeniya et al. [18] applied NSGA-II to the robust optimisation of the CDP considering a varying response time. In reality, vehicles and their ECUs are exposed to temperature differences and similar external factors, which causes the software components to react differently at each invocation. The formulation by Meedeniya et al. [18] treats response time and reliability as probability distributions and presents solutions which are robust with regards to the uncertainty.

In the work by Thiruvady et al. [24], one of the most successful ACO solvers, Ant Colony System (ACS) is combined with constraint programming (CP) to optimise problem instances with different degrees of constrainedness. The constraints considered are limited memory of a hardware unit, collocation restrictions of software components on the same hardware units, and communication between software components. Furthermore, the authors explore the alternative of adding a local search to ACS and CP-ACS. When the search space is extremely constrained, the feasible areas form isolated islands between which the CP solver finds it hard to navigate, unless it is allowed to cross through an infeasible space using relaxation mechanisms. For this reason, ACS outperforms the CP-hybrid in the component deployment optimisation problem, especially when the colocation constraint is very tight.

Both constraints and the objective function affect the suitability of optimisation methods in solving the component deployment problem. The choice of the search operator becomes essential in the efficiency of the optimisation process. In many cases, different search operators may be optimal at different stages of the optimisation process, which motivated this work. Using feedback from the search to adjust the neighbourhood operator has the potential for avoiding getting stuck in a local optimum, or in an infeasible area of the search space.

3 Varying the Neighbourhood Operator for the Component Deployment Problem

Variable neighbourhood search is a general, successful and powerful local search-based method for difficult optimization problems. Local search (LS) based metaheuristics starts from an initial solution and iteratively generates new solutions using a neighbourhood strategy. Each step, a solution that improves over the existing best-so-far solution is chosen. The local search stops when there is no possible improvement, i.e. in a local optimum. Because LS can be stuck in local optima, some advanced local search algorithms consist in alternating (randomly or adaptively) the neighbourhood of the current solution.

The suitability of a local search method for solving an optimisation problem instance depends on the structure of the fitness landscape of that instance. A fitness landscape in the context of combinatorial optimisation problems refers to the (i) search space S, composed of all possible solutions that are connected through (ii) the search operator, which assigns each solution $s \in S$ to a set of neighbours $N(s) \subset S$, and the fitness function $F : S \rightarrow \Re$. As the neighbourhood of a solution depends on the search operator, a given problem can have any number of fitness landscapes. The neighbourhoods can be very large, such as the ones arising from the crossover operator of a genetic algorithm, while a 2-opt operator of a permutation problem has a neighbourhood that is relatively limited in size.

3.1 Neighbourhood strategies

We vary the application of three different neighbourhood operators: OneFlip, kOpt and Perturb. The search starts with a randomly initialised solution, where components are randomly allocated to hardware hosts.

kOpt exchanges the host allocations of k components. In this work, the value of k is equal to 2. Formally, the 2Opt operator produces a new solution d'_i from existing d_i by switching the mapping of two components, e.g. for selected k, l: $d'_i = [d_i(c_1), d_i(c_2), ..., d_i(c_k)..., d_i(c_l), ..., d_i(c_n)]$ while the original solution is $d_i = [d_i(c_1), d_i(c_2), ..., d_i(c_l)..., d_i(c_k), ..., d_i(c_n)]$. With this operator, from one solution d_i we can generate $\binom{n}{2}$ possible new solutions.

OneFlip neighbourhood operator changes the allocation of a single component. Formally, the OneFlip operator produces a new solution d'_i from existing solution d_i by changing the mapping of one components, e.g. for selected k: $d'_i = [d_i(c_1), d'_i(c_2), ..., d_i(c_k), ..., d_i(c_n)]$ while the original parent solution is $d_i = [d_i(c_1), d_i(c_2), ..., d_i(c_k), ..., d_i(c_n)]$. From one solution d_i, we can generate $2n$ new solutions corresponding to the n positions and 2 values for each position.

Perturb changes the allocation of a random component into a random host. The application of this neighbourhood operator creates a new solution d'_i from existing solution d_i by changing the mapping of one components, e.g. for a random k: $d'_i = [d_i(c_1), d'_i(c_2), ..., d_i(c_k), ..., d_i(c_n)]$ while the original parent solution is $d_i = [d_i(c_1), d_i(c_2), ..., d_i(c_k), ..., d_i(c_n)]$. From one solution d_i, we can generate $k_1 n$ new solutions corresponding to the n positions and k_1 values for each position in d_i.

3.2 Adaptive Neighbourhood Search for the Component Deployment Problem

Each operator is applied until a local optimum is found. The solution is then evaluated (line 7 in algorithm 1) and the change in fitness is recorded. The adaptive neighbourhood uses the change in fitness as feedback for adjusting the operator selection probabilities. At the beginning, all operators have equally probability of being selected. The selection probabilities are updated over the iterations based on operator performance. The main steps of these methods are described in algorithm 1. The feedback is used to decide whether to continue to use the current operator or switch to a different one, as shown in algorithm 2.

Algorithm 1 Neighbourhood operators.

```
    procedure OneFlip($S$)
2:      $S^* = S$
        localOptimum = True
4:      for all $c < C$ do
            $h$ = RandomlySelectHost($H$)
6:          $S'$ = AssignComponentToHost($S, c, h$)
            Evaluate($S'$)
8:          if $S' > S^*$ then
                $S^* = S'$
10:         end if
        end for
12:     improvement = FitnessDifference($S, S^*$)
        $S = S^*$
14:     Return(improvement)
    end procedure
16: procedure KOpt($S, k$)
        $S^* = S$
18:     localOptimum = True
        for $c = 0; c < |C| - k; c++$ do
20:         $S'$ = AssignComponentToHost($S, c, d(c+k)$)
            $S'$ = AssignComponentToHost($S, c+k, d(c)$)
22:         Evaluate($S'$)
            if $S' > S^*$ then
24:             $S^* = S'$
            end if
26:     end for
        improvement = FitnessDifference($S, S^*$)
28:     $S = S^*$
        Return(improvement)
30: end procedure
    procedure Perturb($S$)
32:     $c$ = RandomlySelectComponent($C$)
        $h$ = RandomlySelectHost($H$)
34:     $S'$ = AssignComponentToHost($S, c, h$)
        Evaluate($S'$)
36:     if $S' > S^*$ then
            $S^* = S'$
38:     end if
        improvement = FitnessDifference($S, S^*$)
40:     $S = S^*$
        Return(improvement)
42: end procedure
```

The mechanism used for the selection of the neighbourhood operator is a fitness proportionate method (line 4 in algorithm 2). Each operator is assigned a selection probability proportionate to its quality (line 14 in algorithm 2). For the

Algorithm 2 Adaptive neighbourhood search.

```
    procedure AN
2:      $S$ = RandomlyAllocate($C, H$)
        $N$ = SelectNeighbourhoodOperator($P(N)$)
4:      if $N$ == OneFlip then
            $Q(N)$=OneFlip($S$)
6:      end if
        if $N$ == KOpt then
8:          $Q(N)$=KOpt($S, k$)
        end if
10:     if $N$ == Perturb then
            $Q(N)$=Perturb($S$)
12:     end if
        ReportFeedback($Q(N)$)
14:     Return(S)
    end procedure
```

purpose of this work, the fitness change in the solution modified by an operator is used to updates the operator's quality. Given the operator's quality $Q(N)$, the update rule for the operator's selection probability $P(N)$ is calculated as

$$P(N) = \alpha P(N) + (1 - \alpha)Q(N), \tag{7}$$

where α is a parameter that controls the influence of previous versus immediate performance. Higher values for α increase the effect of previous performance, whereas lower values focus on immediate effects. In this study, α was set to 0.9. Formally, the operators quality is calculated as:

$$Q(N) = \frac{|f(S) - f(S^*)|}{f(S)} \tag{8}$$

where $f(S)$ is the quality of the solution at the start of the search, and $f(S^*)$ is the quality of the optimised solution.

The deterministic neighbourhood strategy, performs a variable neighbourhood search by applying the three neighbourhood operators sequentially, in the order given in Algorithm 3. We have selected this order arbitrarily and can be changed by the user. Another alternative would have been to select at random one of these operators.

The variable neighbourhood search, on the other hand, assigns equal probabilities to OneFlip and kOpt. The main steps are listed in algorithm 3. At each iteration, VN selects one of the two operators with equal probability. The Perturb operator is the most disruptive operator since it can make the largest changes in the search space. Therefore, Perturb is applied after the OneFlip or kOpt with a very small probability, which ensures that the search does not get trapped in a local optimum.

Algorithm 3 Deterministic and variable neighbourhood search operators.

```
procedure DN
    $S$ = RandomlyAllocate($C, H$)
    OneFlip($S$)
    Perturb($S$)
    KOpt($S, k$)
    Return(S)
end procedure

procedure VN
    $S$ = RandomlyAllocate($C, H$)
    r = Random([0,1])
    if $r > 0.5$ then
        OneFlip($S$)
    end if
    if $r < 0.5$ then
        KOpt($S, k$)
    end if
    p = Random([0,1])
    if $p < 0.01$ then
        Perturb($S$)
    end if
    Return(S)
end procedure
```

4 Experiments

To evaluate the efficiency of the proposed methods for varying the neighbourhood search operator, we have designed a set of experiments with problems of different sizes and level of difficulty. The design of experiments and the analysis of results is described in the following sections.

4.1 Experimental Design

The problems used for the experiments consist of a number of randomly generated instances with varying complexity and constrainedness. The memory constraint takes its tightness from the ratio of components to hardware hosts - the fewer hosts, the less ‘space’ there is for components. The smallest instances consist of 10 hosts and 23 software components whereas the largest instances consist of 62 hardware hosts and 130 software components. All algorithms were allowed 50 000 function evaluations, and all trials were repeated 30 times to account for the stochastic behaviour of the algorithms. Results from the 30 runs were analysed and compared using the Kolmogorov-Smirnov (KS) non-parametric test [22]. Furthermore, the effect size was reported for each experiment.

The validity of the presented experiments may be questioned on the grounds that the results may only reflect the performance of the algorithms in certain problem instances, and there is a chance that the approaches may perform differently for other problems. In the design of experiment, we aimed at reducing this threat by generating problem instances of different sizes and characteristics. Instead of manually setting specific problem properties, we developed a problem generator integrated in ArcheOpterix [2]. The problem generator and the problem files can be downloaded from `users.monash.edu.au/~aldeidaa/ArcheOpterix.html`. As a result, the experiments set themselves apart from an instance-specific setting to a broader applicability.

4.2 Results

The experiments were performed on a 64-core 2.26 GHz processor computer. There was little difference in the run-times of the different optimisation schemes for the same problem instances. The main difference in run-time was observed between the problem instances with different size. Solving the smaller instances was faster, since the evaluation of the quality attributes takes less time. The distributions of the fitness values (reliability function in eq. 3) of the 30 runs are visualised as boxplots for Adaptive Neighbourhood (AN), Variable neighbourhood (VN), deterministic neighbourhood (DN) and local search (LS). The middle lines represent the median value for each case. The means and standard deviations for all problem instances and optimisation schemes are shown in Table 1. AN outperforms the other strategies in the majority of the problem instances, which indicates that using feedback from the search to adapt the neighbourhood operator benefit the optimisation process.

Table 1: The mean and standard deviation of the 30 trials of adaptive neighbourhood search (AN), variable neighbourhood search (VN), deterministic neighbourhood search (DN) and local search (LS).

	AN		**VN**		**DN**		**LS**	
Problem	Mean	Std. dev.	Mean	Std. dev.	Mean	Std. dev.	Mean	Std. dev.
H10C23	0.999871	0.000000	0.999871	0.000000	0.998980	0.000057	0.999871	0.000000
H20C45	0.999993	0.000000	0.999979	0.000004	0.999965	0.000009	0.999993	0.000000
H25C54	**0.999981**	0.000000	0.999980	0.000000	0.999935	0.000010	0.999979	0.000000
H30C65	**0.999780**	0.000002	0.999746	0.000007	0.999615	0.000048	0.999743	0.000005
H35C74	**0.999986**	0.000000	0.999980	0.000001	0.999912	0.000023	0.999983	0.000000
H42C85	**0.999777**	0.000039	0.999770	0.000027	0.999759	0.000037	0.999737	0.000045
H55C107	**0.999796**	0.000004	0.999771	0.00007	0.999679	0.000006	0.999667	0.000043
H62C130	**0.999974**	0.000009	0.999932	0.000008	0.999918	0.000016	0.999919	0.000013

As the adaptive neighbourhood strategy consistently outperforms the three other optimisation schemes, we use the Kolmogorov-Smirnov (KS) non-parametric test [22] to check for a statistical difference. The 30 results of the repeated trials

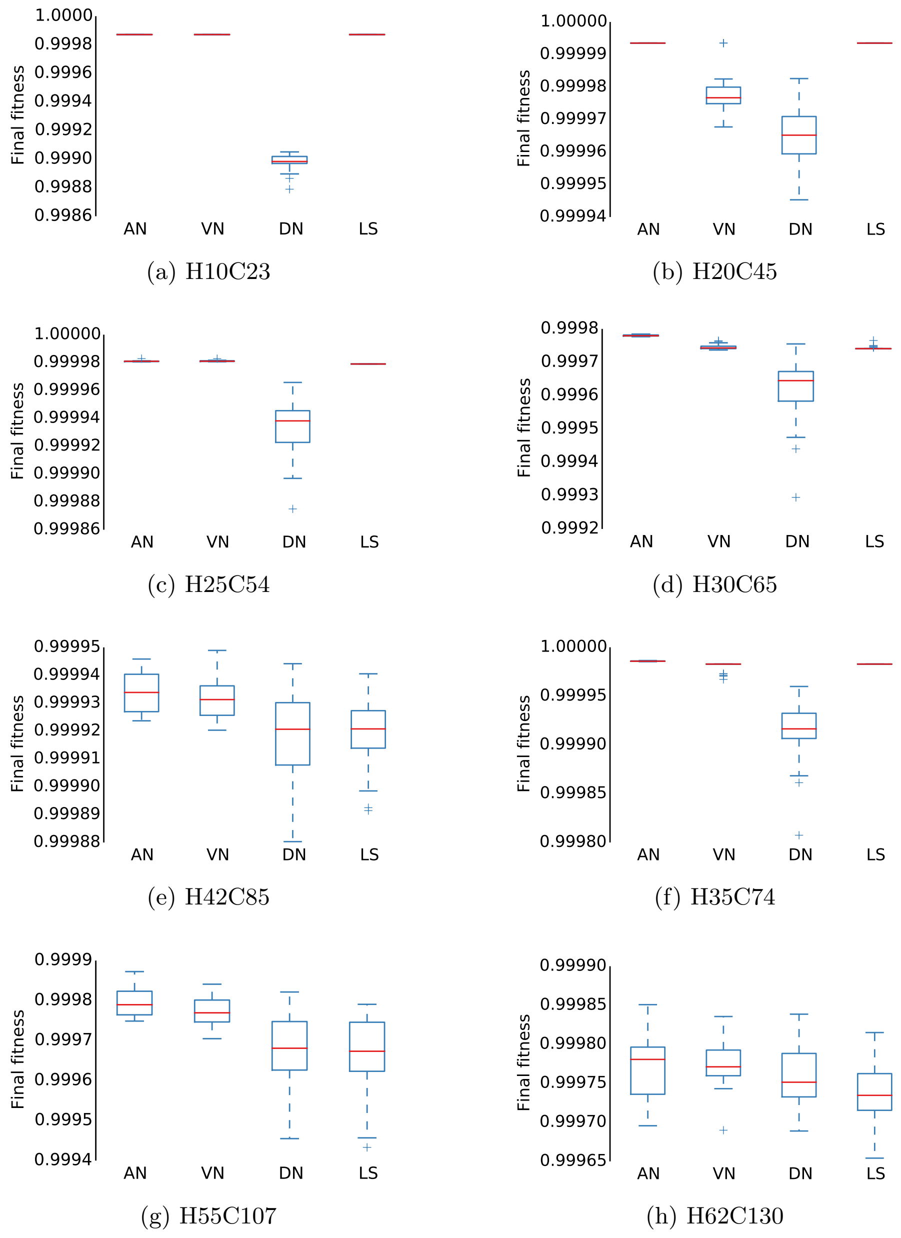

Fig. 1: Boxplots of fitness values (reliability function in eq. 3) for Adaptive Neighbourhood (AN), Variable neighbourhood (VN), deterministic neighbourhood (DN) and local search (LS).

Table 2: The KS test values and effect size of the 30 trials of adaptive neighbourhood search vs. variable neighbourhood search (AN vs. VN), adaptive neighbourhood search vs. deterministic neighbourhood search (AN vs. DN) and adaptive neighbourhood search vs. local search (AN vs. LS).

	KS test			**Effect size**		
Problem	AN vs. VN	AN vs. DN	AN vs. LS	AN vs. VN	AN vs. DN	AN vs. LS
H10C23	1	≤ 0.01	1	0.00	0.99	0.00
H20C45	≤ 0.01	≤ 0.01	1	0.92	0.91	0.00
H25C54	0.02	≤ 0.01	≤ 0.01	0.99	0.95	0.99
H30C65	≤ 0.01	0.05	≤ 0.01	0.95	0.92	0.98
H35C74	≤ 0.01	≤ 0.01	≤ 0.01	0.97	0.98	0.99
H42C85	0.01	0.03	0.03	0.91	0.92	0.94
H55C107	≤ 0.01	0.02	≤ 0.01	0.90	0.99	0.90
H62C130	0.01	0.04	0.05	0.92	0.90	0.93

for each of the problem instances were submitted to the KS analysis. The adaptive neighbourhood search (AN) was compared to the other three optimisation schemes, with a null hypothesis of an insignificant difference between the performances (AN vs. VN, AN vs. DN, and AN vs. LS). The results of the tests are shown in Table 2.

All KS tests, used for establishing that there is no difference between independent datasets under the assumption that they are not normally distributed, result in a rejection of the null hypothesis at a 95% confidence level in the majority of the cases. AN and LS have the same performance in two problems H20C45 and H10C23. The search space of these two instances is relatively small, and the problems can be solved to high quality with local search. For larger and more complex search spaces, the local search method fails at finding good results. The deterministic neighbourhood search method has the worst performance out of the four optimisation schemes. Clearly, applying the three operators sequentially does not benefit the search, although for the large problem instances, varying the neighbourhood produces better results than local search, even if it is done in a deterministic way.

Since statistical significance depends on the sample size, we also compute the effect size for each comparison (AN vs. VN, AN vs. DN, and AN vs. LS), as shown in Table 2. The effect of the sample size is measured using the Cohen's d estimation [8], which considers the pooled standard deviation. In reporting the effect size, we follow the guidelines proposed by Cohen [8]: a 'small' effect size is 0.2, a 'medium' effect size is 0.5, and a 'large' effect size is 0.8. In essence, the effect size indicates the number of standard deviations difference between the means of the samples. The effect size for the problem instances that show statistical significance in terms of the KS test was above 0.9. As a results, it can be concluded that the difference in the performance of the optimisation schemes is meaningful, and that the AN strategy is the most successful search method for the component deployment problem.

5 Conclusion

This paper propose an adaptive variable neighbourhood search for component deployment components that uses multiple neighbourhood operators to escape local optimum. These operators are adaptively selected such that the operator that improves the most the fitness value and most frequently the current solution is selected the most often. Two other versions of this variable neighbourhood search algorithm alternate uniformly at random or deterministically the three operators. We test the three proposed algorithms and a simple version of multiple restarts local search on several instances of component deployment problems with different level of difficulty. The experimental results show that the adaptive variable neighbourhood algorithm outperforms the other algorithms. We conclude that the local search algorithms are useful optimization algorithms for component deployment problems.

Acknowledgements: This research was supported under Australian Research Council's Discovery Projects funding scheme, project number DE 140100017.

References

1. Aldeida Aleti. Designing automotive embedded systems with adaptive genetic algorithms. *Automated Software Engineering*, pages 1–42, 2014.
2. Aldeida Aleti, Stefan Björnander, Lars Grunske, and Indika Meedeniya. ArcheOpterix: An extendable tool for architecture optimization of AADL models. In *Model-based Methodologies for Pervasive and Embedded Software*, pages 61–71. ACM and IEEE Digital Libraries, 2009.
3. Aldeida Aleti, Barbora Buhnova, Lars Grunske, Anne Koziolek, and Indika Meedeniya. Software architecture optimization methods: A systematic literature review. *IEEE Transactions on Software Engineering*, 39(5):658–683, 2013.
4. Aldeida Aleti and Lars Grunske. Test data generation with a kalman filter-based adaptive genetic algorithm. *Journal of Systems and Software*, 103(0):343 – 352, 2015.
5. Aldeida Aleti, Lars Grunske, Indika Meedeniya, and Irene Moser. Let the ants deploy your software - an ACO based deployment optimisation strategy. In *ASE*, pages 505–509. IEEE Computer Society, 2009.
6. Aldeida Aleti and Indika Meedeniya. Component deployment optimisation with bayesian learning. In *ACM Sigsoft symposium on Component based software engineering*, pages 11–20. ACM, 2011.
7. Ismail Assayad, Alain Girault, and Hamoudi Kalla. A bi-criteria scheduling heuristic for distributed embedded systems under reliability and real-time constraints. In *Dependable Systems and Networks*, pages 347–356. IEEE Computer Society, 2004.
8. J. Cohen. *Statistical power analysis for the behavioral sciences.* 1988.
9. David W. Coit and Abdullah Konak. Multiple weighted objectives heuristic for the redundancy allocation problem. *IEEE Transactions on Reliability*, 55(3):551–558, 2006.

10. Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A Fast Elitist Multi-Objective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6:182–197, 2000.
11. Carlos M Fonseca, Peter J Fleming, et al. Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization. In *ICGA*, volume 93, pages 416–423, 1993.
12. Michael Guntsch and Martin Middendorf. Solving multi-criteria optimization problems with population-based aco. In *Evolutionary Multi-Criterion Optimization. Second International Conference, EMO 2003*, pages 464–478. Springer, 2003.
13. Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, 45(1):11:1–11:61, December 2012.
14. Mark Harman and Phil McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transaction Software Engineering*, 36(2):226–247, 2010.
15. ISO/IEC. IEEE international standard 1471 2000 - systems and software engineering - recommended practice for architectural description of software-intensive systems, 2000.
16. Peter Kubat. Assessing reliability of modular software. *Operations Research Letters*, 8(1):35–41, 1989.
17. Sam Malek, Nenad Medvidovic, and Marija Mikic-Rakic. An extensible framework for improving a distributed software system's deployment architecture. *IEEE Transactions on Software Engineering*, 38(1):73–100, 2012.
18. Indika Meedeniya, Aldeida Aleti, Iman Avazpour, and Ayman Amin. Robust archeopterix: Architecture optimization of embedded systems under uncertainty. In *Software Engineering for Embedded Systems*, pages 23–29. IEEE, 2012.
19. Indika Meedeniya, Aldeida Aleti, and Lars Grunske. Architecture-driven reliability optimization with uncertain model parameters. *Journal of Systems and Software*, 85(10):2340–2355, 2012.
20. Indika Meedeniya, Barbora Buhnova, Aldeida Aleti, and Lars Grunske. Architecture-Driven Reliability and Energy Optimization for Complex Embedded Systems. In *Research into Practice - Reality and Gaps, 6th International Conference on the Quality of Software Architectures*, pages 52–67. Springer, 2010.
21. Indika Meedeniya, Barbora Buhnova, Aldeida Aleti, and Lars Grunske. Reliability-Driven Deployment Optimization for Embedded Systems. *Journal of Systems and Software*, 2011.
22. A. N. Pettitt and M. A. Stephens. The kolmogorov-smirnov goodness-of-fit statistic with discrete and grouped data. *Technometrics*, 19(2):205–210, 1977.
23. Songqing Shan and G. Gary Wang. Reliable design space and complete single-loop reliability-based design optimization. *Reliability Engineering and System Safety*, 93(8):1218 – 1230, 2008.
24. Dhananjay Thiruvady, I Moser, Aldeida Aleti, and Asef Nazari. Constraint programming and ant colony system for the component deployment problem. *Procedia Computer Science*, 29:1937–1947, 2014.
25. Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. Automatic program repair with evolutionary computation. *Communications of the ACM*, 53(5):109–116, 2010.
26. David H. Wolpert and William G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.