A Model-Based Genetic Algorithm Framework for Constrained Optimisation Problems

Authors

Affiliations,
`Emails`

Abstract. Two major challenges are presented when applying genetic algorithms (GAs) to constrained optimisation problems: modelling the problem, and handling the constraints of the problem. The field of constraint programming (CP) has enjoyed extensive research in both of these areas. CP frameworks have been devised which allow arbitrary problems to be readily modelled, and their constraints handled efficiently. Our work aims to combine the modelling and constraint handling of a state-of-the-art CP framework with the efficient population-based search of a GA. We present a new general hybrid CP / GA framework which can be used to solve any constrained optimisation problem that can be expressed using the language of constraints. The framework is presented in its most basic form, and much remains to be drawn from both the GA and CP literature to further improve it. The efficacy of this framework as a general heuristic for constrained optimisation problems is demonstrated through experimental results on a variety of classical combinatorial optimisation problems commonly found in the literature.

Keywords: Genetic Algorithms, Hybridization, Constraint Handling, Combinatorial Optimization, Modelling, Meta-heuristics

1 Introduction

Genetic algorithms (GAs, [16, 3]) are a family of optimisation meta-heuristics based on the evolutionary principle of *survival of the fittest*. The way GAs solve an optimisation problem is by evolving a *population* of candidate solutions (*individuals*). A solution is typically represented as a sequence of bits (a *genotype*), and is associated with a *fitness* function that reflects the quality of the solution. The fitness measure is usually calculated based on a realisation of the genotype, i.e., a *phenotype*, which is a representation of the solution in its original domain. The population of solutions is *evolved* by using operators (crossover and selection) drawn from genetics. The goal of these operators is to recombine the fittest individuals in the population (*parents*) to create the next generation of candidate solutions (*offspring*). The idea is that by selecting the fittest individuals and recombining them, the overall fitness will increase with time. Most GAs also include a mutation operator, which slightly modifies the offspring to increase the diversity of the genetic pool. In the context of a GA, modelling a problem involves devising an effective encoding for the solutions. This can be challenging, particularly in the presence of hard constraints, that can make a solution worthless if not satisfied. Handling hard constraints typically requires developing problem-specific and effective crossover and mutation operators. Despite these challenges, GAs are a popular technique for solving difficult optimisation problems, and have been successful in applications ranging from logistics [23] to renewable energy [25].

Constraint programming (CP, [15]) is a paradigm for modelling and solving constraint satisfaction problems (CSPs) and constrained optimisation problems (COPs). In CP, a problem is modelled in terms of decision variables and constraints. Decision variables are associated with initial *domains*, i.e., finite sets of values that define the search space of the problem (e.g., $x \in \{1 \dots 8\}, y \in \{3 \dots 11\}$). Constraints represent logic relations between variables and define the allowed set of values that the variables can take (e.g., $x \geq y$, or $y < 7$). Operationally, constraints are implemented by *propagators*, filtering algorithms that remove from the domains of the variables the values that cannot possibly be part of a solution, enforcing various notions of *consistency*. For example, the propagation of the above constraints would yield $x \in \{3 \dots 8\}, y \in \{3 \dots 6\}$. More complex constraints exist, e.g., **alldifferent**$(x_1, \dots, x_n)$ enforces that the variables $x_1, \dots, x_n$ assume pairwise different values. Sophisticated propagators exist for such constraints, which allow complex problem structures to be captured in a very concise way. Problems are solved using some form of backtracking tree-search that interleaves *branching* and constraint *propagation* steps. At each node of the tree-search *variable and value selection heuristics* are used to select the next variable to assign, and the value to assign to it. Then, the propagators of the constraints that involve the selected variable are triggered, and proceed to filter the domains of the other variables according to their logic. This in turn can trigger the propagation of other constraints, and the algorithm continues until a fixpoint is reached and the domains are all consistent with the set of constraints. If the propagation fails, the algorithm backtracks to a previous decision point and the search proceeds from there. Customarily, CP-style optimisation is complete: it proceeds until the domains of the variables are singletons that satisfy all the constraints, or it is proven that there is no solution to the problem. CP has had significant success in dealing with challenging constraints in areas including routing and scheduling. Modern CP frameworks like ILOG CPLEX CP OPTIMIZER, MINIZINC, CHOCO, OSCAR, OR-TOOLS, or GECODE provide large libraries of global constraints with efficient propagators, making modelling COPs nearly trivial. However, research in CP has focused largely on finding feasible solutions, rather than near-optimal solutions. Constrained optimisation problems are typically solved by imposing an upper bound on the cost of the next solution every time a new best solution is found.

The goal of this work is to propose a hybrid framework to combine genetic algorithms with constraint programming. By bringing together these orthogonal technologies, we hope to leverage the modelling and constraint handling capabilities of CP, and the efficient, parallel, population-based search strategies of GAs.

We hope that this framework will serve as a starting point for further research into combining the numerous advantages of these two worlds.

We validate our contribution by showing the applicability of our approach by running an experimental analysis on a number of classical combinatorial optimisation problems. The outcome of the experiments allows us to clarify the strengths and weaknesses of this hybridisation, and identify future research directions.

1.1 Related Work

In this section we briefly summarise the existing literature on the integration of CP and GAs and between CP and other meta-heuristics. The latter class of approaches has been explored more than the former, and some of the proposed approaches have proven to be both effective and widely applicable, two characteristics that we seek to obtain in our work.

Constraint programming has been used within GAs to reject infeasible individuals [9], which can save computational effort by avoiding working on infeasible individuals, however it can be very inefficient if individuals are rejected too often. CP has also been used within GAs to repair infeasible individuals [5, 24], both as part of the evaluation and as a standalone *repair* operator. This approach allows GAs to more effectively handle hard constraints, however repairing an individual can sometimes be difficult, and the repaired individual might be far from the original one.

There has been more significant work on other approaches hybridising CP and meta-heuristics [10]. The most prominent example is certainly CP-based large neighbourhood search (LNS, [19, 14, 11]). LNS is a local search meta-heuristic that uses constraint propagation as a way to reduce the size of a solution's neighbourhood by filtering away infeasible solutions. LNS proceeds by first finding an initial solution, then repeatedly destroying part of a solution, and re-optimising it. When used in conjunction with propagation techniques, the relaxed variables are usually re-optimised through *branch & bound* (B&B). LNS has been successfully applied to a range of real-world applications, including routing [8] and scheduling [12] problems. Ant colony optimisation (ACO) has also been hybridised with CP [10] by using ACO in the ordering heuristics to guide the branching of a backtracking B&B search. This has had encouraging results when compared to the default ordering heuristics of ILOG CPLEX CP OPTIMIZER [7], however it is a complete algorithm and does not scale well on real-world problems.

Our contribution belongs to the first class of approaches, however the characteristics of our framework are similar to the ones of LNS: a model-based approach that only assumes the availability of an underlying constraint model. A key advantage that we seek to bring in from genetic algorithms is the ability to explore large and complex search spaces without getting easily trapped in local minima.

2 Our framework

Our approach retains the overall logical structure of a classic GA (see Algorithm 1 for reference), with the difference that, at all stages of the search, each individual in the population is a *feasible* solution of a constraint model of the problem being solved. By this we do not only mean that each individual is always a feasible solution of the problem, but also that it is implemented *in practice* as a CP solution, rather than as a string of values as is customary in GAs. The feasibility of the individuals is operationally guaranteed at all times by enforcing constraint propagation whenever a new individual is created or an existing solution is modified.

Algorithm 1 Classical Genetic Algorithm

```
Require: Population size: $\mu$
  procedure GeneticAlgorithm
      $P_I \leftarrow$Initialiser($\mu$)                  ▷ $P_I$ is the incumbent population
      while stopping criteria is not met do
          Clear($P_O$)                               ▷ $P_O$ is the offspring population
          while $|P_O| < \mu$ do
              $I_1, I_2 =$Select($P_I$)
              $P_O \leftarrow$Crossover($I_1, I_2$)
          end while
          Mutate($P_O$)
          Evaluate($P_O$)
          $P_I =$ Elitism($P_I, P_O$)
      end while
      return Best($P_I$)                            ▷ Return the fittest individual
  end procedure
```

In this section, we will use this structure in Algorithm 1 as a blueprint, and describe how the various components can be specialised to reflect and leverage the existence of an underlying constraint model and a propagation engine.

2.1 Initialiser

The first step of any genetic algorithm is to initialise a population of solutions. In our approach, we use a model-based Initialiser routine. The routine first creates μ unassigned CP solutions, i.e., solutions in which the domains of the decision variables are set to their original ranges. Then, each solution is built through a depth-first tree-search in which the next variable to be assigned is chosen as the one with the highest *accumulated failure count* (AFC[1]), and the

[1] The accumulated failure count variable selection strategy [17] corresponds to a dynamic randomised version of the *fail first* heuristic, and is the default variable selection for some constraint programming solvers.

value to assign to it is chosen uniformly at random. Constraint propagation is enforced at each step of the tree-search, so that the initial population consists only of feasible solutions.

Implementation details. Because of the random variable and value selection heuristics, the generated solutions will likely be different. To make sure that this is the case, every time a new solution is created, we run a uniqueness test against the existing population. If the test fails, we generate a new solution from scratch. If, after a certain number of attempts, the procedure has not succeeded in generating a new (and diverse) individual, we settle on a smaller population.

2.2 Selection

Since the selection of parents does not modify or generate new individuals, any operator from the GA literature can be used. In our current implementation, we use a fairly standard *roulette wheel* [6] selection mechanism, in which the parents are chosen proportionally to their fitness. This increases the probability of transferring good solution components into the next generation.

2.3 Crossover

Once two parents have been selected, we combine them to obtain a new offspring. The two goals that we want to achieve in the crossover are *i)* to make the offspring inherit genetic material from its parents, and *ii)* to ensure that the offspring is feasible. To achieve this, we initialise the offspring with an empty CP solution, and then assign its variables using a depth-first tree-search which takes the two parents as inputs. In particular, while the variable selection heuristic chooses the next variable v to assign uniformly at random, the value selection heuristic biases the search towards the values of the variable v in the parents. In particular, the values to assign to v are attempted in one of the following orders

$$ord_1 = \langle val(v_1), val(v_2), \text{values in } dom(v) \text{ in random order}\rangle$$
$$ord_2 = \langle val(v_2), val(v_1), \text{values in } dom(v) \text{ in random order}\rangle$$

where $val(v_x)$ represents the value of the variable v in the parent x, $dom(v)$ represents the domain of v in the offspring, and the orders ord_1 and ord_2 are chosen with equal probability. In other words, the variable v in the offspring will have, with high probability, the value of the corresponding variable in either of the two parents, and with lower probability a different value from its own domain. This satisfies the first goal. Since constraint propagation is enforced at each step of the tree-search, the new generation can only consist of feasible solutions. This satisfies the second goal.

Implementation details. The reason we allow offspring to take values that do not belong to any of the parents is that, after assigning the first $n-1$ variables, propagation may have filtered these values from domain of the n^{th} variable.

In this case, the backtracking tree-search would still succeed in producing an offspring, but it might not strictly represent a recombination of the parents.

In order to leverage the presence of a propagation engine, we also constrain the fitness of the new offspring to be at least as good as the least fit parent. In doing this, we are trying to direct the search away from low-fitness areas of the search space.

2.4 Mutation

After a new offspring is generated, we randomly mutate each of its variables with a *mutation probability* (p_m), a parameter of the solver. An individual is mutated by relaxing, i.e., setting to their original domain, individual variables and then using a depth-first tree-search to assign (potentially different) values to those variables. Similarly to the initialiser, during the tree-search the variables are sorted by accumulated failure count, and the values are selected randomly from their domains. Since this random value ordering can lead to poor quality solutions, we constrain the obtained individual to be at least as fit as it was before mutation. We allow the resulting offspring to be exactly the same as before the mutation.

2.5 Elitism

Elitism is a mechanism to retain the best solutions in the resulting population, and does not modify individuals themselves. As such, and similar to selection, any approach from the GA literature can be used. In our current implementation, we use an approach that retains elite individuals across iterations. To achieve this, an *exempt fraction* (f_e) of the fittest individuals in the incumbent population is copied verbatim to the new population. The copied individuals can be still taken into consideration for crossover.

2.6 Time-limited tree-search

When using tree-search as a sub-procedure to repair a solution or generate a new one, wrong decisions towards the root of the search tree can cause backtracking to take a long time to find a feasible solution. From experience, we learnt that in these situations it is better to stop the search and restart it. In all of the procedures described above, we use a time limit of 5 milliseconds per decision variable for each tree search, after which we restart it. In principle, this value is a parameter of the solver, and should be chosen by parameter tuning. In practice, in our experience, 5 milliseconds is a rather robust value for most applications.

3 Experimental analysis

We conduct an experimental analysis to demonstrate the general applicability of our CP / GA framework (ConGA) to solve a variety of combinatorial optimisation problems. To this end, we model four classical combinatorial optimisation problems in the Gecode constraint programming framework, and compare the performance of our approach with the off-the-shelf B&B tree-search strategy provided by Gecode[2], and an established CP-based large neighbourhood search (LNS) strategy developed as a freely available Gecode plug-in (Gecode-LNS).

Because of space constraints, we are unable to extensively describe the specific LNS procedure strategy used here; it suffices to say that we repeatedly relax a fraction δ (initially $\delta = \delta_{init} = 2$) of the decision variables of the incumbent solution, and then we re-optimise them through a B&B tree-search. The B&B search has a limited time budget $n \cdot t_{var}$ where n is the number of relaxed variables, and the procedure is repeated until until a stopping condition is met. The search strategy is slightly adaptive, in that the fraction of relaxed variables is increased by one if the repair step fails to find a feasible solution within ii_{max} iterations, and reset to δ_{init} when / if a solution is finally found. In the repair step, the cost of the repaired solution is constrained to be better or at least as good as that of the incumbent solution which, in combination with the constraint propagation, drives the search towards low-cost areas of the search space. Notably, this LNS strategy has requirements comparable to those of our framework in that, in its simplest use case, it only requires to be initialised using a CP model. For a more extensive discussion of the approach, we refer the reader to [21].

In the following, we first briefly describe our benchmark problem domains and the selected instances. Then we specify our experimental setup in terms of hardware, software, and parameter settings for both ConGA and LNS. Finally, we report and discuss our results.

3.1 Problem domains

In the spirit of testing the generality of our approach, we deliberately choose four classical combinatorial optimisation problems. Below, we briefly describe how we modelled them with constraints.

TSP. A travelling salesperson needs to visit n cities. The associated optimisation problem consists of finding the most efficient way to visit each city exactly once, and return to the starting city. We model this by using a decision variable s_i (the *successor* of i) for each city i, representing the city the salesperson will travel to after city i . We use a global **circuit** [18] constraint to ensure that our successor variables form a Hamiltonian circuit. Our objective is to minimise the total distance covered by this circuit. In Gecode the total distance is propagated for free by the circuit constraint implementation.

The TSP instances we use for our benchmark are chosen arbitrarily from the literature [13] to represent a range of problem sizes for which a feasible initial solution can typically be found within the time budget. These instances are listed in Table 1 which reports, along with the instance name, the number of cities.

[2] Branch & bound tree-search is also the default search strategy provided by most constraint programming solvers.

Instance	Dimension (n)
berlin52	52
kroA100	100
bier127	127
a280	280
lin380	380

Table 1: TSP instances for our benchmark.

CVRP. In the capacitated vehicle routing problem, a fleet of k vehicles of identical capacity Q must serve m customers with different demands $d_i, i \in \{1 \ldots m\}$ for a generic commodity. The cost to travel between any two locations (depot included) i, j is encoded by a cost function C_{ij}. The goal is to find the cheapest way to serve each customer's demand without overloading the vehicles.

The model for this problem can be seen as an extension of the TSP model, in that the set of routes are represented by a single long circuit. At each step we add *load* variables to reflect the amount of commodity loaded and unloaded at each customer, and bound these by the vehicle capacity Q. Moreover, we introduce k nominal depots (one for each vehicle) as a modelling convenience, so that we can treat the routes independently. To demonstrate the simplicity of modelling with constraint programming, this model is implemented with MiniZinc in Figure 1. We also model a version of the CVRP where the capacity constraints are made soft by unbounding the domains of the load variables and introducing slack variables. Each unit of excess load has a fixed penalty of 10^4 that adds to the cost of the solution.

The CVRP instances used for benchmarking are chosen arbitrarily from the literature [20] to represent a range of problem sizes for which a feasible initial solution can typically be found within the time budget. These instances are listed in Table 2.

Instance	Customers (m)	Vehicles (k)
E-n23-k3	22	3
A-n32-k5	31	5
B-n34-k5	33	5
B-n44-k7	43	7
F-n45-k4	44	4
E-n76-k7	75	7
M-n121-k7	120	7
M-n200-k17	199	17

Table 2: CVRP Instances

JSSP. In the job shop scheduling problem we have j jobs and m machines. Each job consists of m tasks, each with a certain duration, which must be completed in a specified order on specific machines. Each machine can only work on one task at a time, and each job can only have one task being worked on at a time.

```
% Input data
int: NVehicles;
int: NCustomers;
int: Capacity;
set of int: InLocs = 1..NCustomers+1;
array[InLocs] of int: Demands;
array[InLocs,InLocs] of int: Costs; % cost matrix

% Introducing a nominal depot for each vehicle
int: NLocs = NVehicles + NCustomers;
set of int: Locs = 1..NLocs;
set of int: Depots = 1..NVehicles;
set of int: Customers = NVehicles+1..NLocs;

% Variables
array[Locs] of var Locs: succ;
array[Locs] of var 0..Capacity: load;
var int: total_cost; % total cost of routes

% Circuit constraint
constraint circuit(succ); % Enforce hamiltonian circuit

% Load constraints
constraint forall(i in Locs) (
 if succ[i] in Customers then
  load[succ[i]] = load[i] + Demands[succ[i]-NVehicles+1]
 else
  load[succ[i]] = 0
 endif
);

% Cost constraint
constraint total_cost = sum (i in Locs) (
 let {
  int: mod_i = if i in Depots then 1
   else i - NVehicles + 1 endif,
  var int: mod_succ = if succ[i] in Depots then 1
   else succ[i] - NVehicles + 1 endif
 } in Costs[mod_i, mod_succ]
);

% Redundant constraint ensuring no depot follows a depot
constraint forall (i, j in Depots) (succ[i] != j);
```

Fig. 1: CVRP-hard model in MiniZinc.

We need to schedule these tasks on the machines such that the total time taken, or *makespan*, is minimised. To model this, we have a decision variable for each task representing its start time, and a variable for each task representing its end time. The end time of each task is simply constrained to be the sum of the start time of that task and its duration. Global **unary** resource scheduling constraints are used to ensure that each machine only processes one task at a time, and each job only has one task processed at a time. To guarantee the tasks are completed in the specified order, we enforce for each task that its start time is greater than its predecessor's end time. Finally, our objective is the makespan which is constrained to be the latest end time.

In the soft version of this model, we relax the precedence constraints by introducing slack variables, each representing how much earlier a task starts than its predecessor. This is penalised with a fixed penalty of 10^3 for each unit of time a task starts before its predecessor.

The JSSP instances used for benchmarking are chosen arbitrarily from the literature [1] to represent a range of problem sizes for which a feasible initial solution can typically be found within the time budget. These instances are listed in Table 3.

Instance	Jobs (j)	Machines (m)
orb08	10	10
ft20	20	5
abz8	20	15
la31	30	10
swv20	50	10

Table 3: JSSP Instances

BPP. In the bin packing problem, there are n items, each with a certain weight. We have bins of a fixed capacity Q in which we need to pack the items. The goal is to pack these bins such that as few as possible bins are used, without overloading any bin. This is modelled with a decision variable for each item representing which bin it is placed into, and load variables for each bin. The global bin packing constraint is used to ensure that the load variables are indeed the load of each bin. It is impossible to overload any bin as the upper bound of each load variable is the capacity. Our objective is the number of bins used, which is provided by a global constraint counting the number of nonzero load variables. In the soft version of this model, the upper bound on the load variables is removed and slack variables introduced. Each unit of excess load has a fixed penalty of 10.

The BPP instances used for benchmarking are chosen arbitrarily from the literature [4] to represent a range of problem sizes for which a feasible initial solution can typically be found within the time budget. These instances are listed in Table 4.

Instance	Items (n)
Falkenauer_u120_09	120
Hard28_BPP14	160
N1W2B3R7	50
N2W2B2R4	100
Schwerin2_BPP25	120

Table 4: BPP Instances

3.2 Experimental setup

All of the compared approaches are implemented in C++ (GCC 6.1.0), and based on the GECODE 5.0.0 constraint programming framework. All of the experiments are run in single-threaded mode on a cluster with INTEL® XEON® E5-2660v3 processors operating at 2.6 GHz.

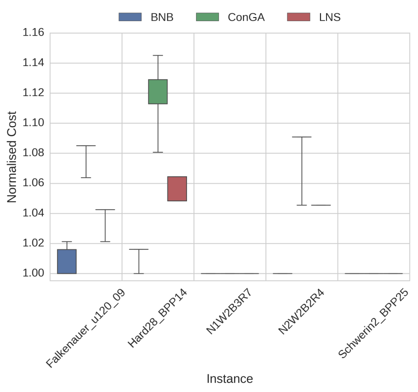
The B&B solver does not require any additional parameters to run, however it requires specifying variable and value selection strategies. At each branching step we choose the variable with the highest AFC first, and we assign (or forbid) a random value in its domain.

Unlike B&B, LNS and CONGA require setting a number of parameters. For this benchmark, we do not perform an extensive tuning of the approaches, because the goal is to show the viability of our framework, rather than delivering a finely configured piece of software. For LNS we set the parameters, described in [21], according to our previous experience with the software. In particular, the configuration used to obtain the below results is $\delta_{init} = 2, \delta_{max} = 5, ii_{max} = 20$, and $t_{var} = 5ms$, and the cost of the new solution is constrained to be at least as good as the one of the incumbent. For CONGA we set the parameters based on previous experiments during the development phase. It is understood that a proper tuning of the framework is fundamental to achieve reasonable performance; this will be the focus of future phases of the project. The parameter configuration used for CONGA to obtain the below results is $f_e = 0.25, p_m = 0.1$ and $\mu = 10$.

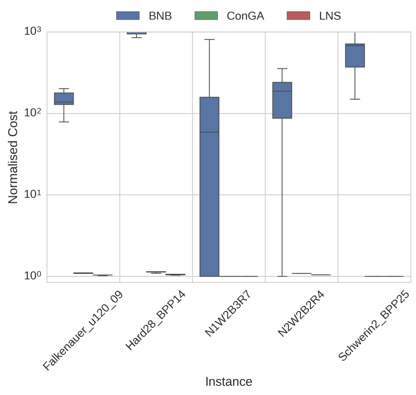
3.3 Results

Figures 2a, 2b, 3a, 3b, 4a, 4b, and 5 compare the performance of B&B, LNS, and CONGA on the bin packing problem (hard and soft variant), the capacitated vehicle routing problem (hard and soft variant), the job shop scheduling problem (hard and soft variant) and the travelling salesperson problem, respectively. Each experiment was given a time budget of 600 seconds and was repeated 10 times.

In the bin packing domain (Figs. 2a and 2b) we observe some significant differences in the performance patterns between the hard and the soft variants of the model. In the soft variant, both CONGA and LNS vastly outperform B&B, and LNS outperforms CONGA on all instances except the one that is very easy, in which all methods reach the same value. Conversely, in the hard variant, B&B consistently outperforms both LNS and CONGA. Our intuition is that, in the hard variant, B&B is able to exploit the higher propagation strength and longer tree-search to prune the search space and reach the optimal



(a) Hard variant.

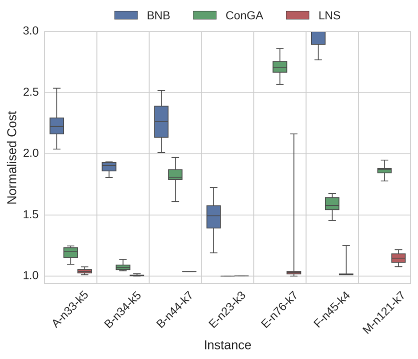


(b) Soft variant.

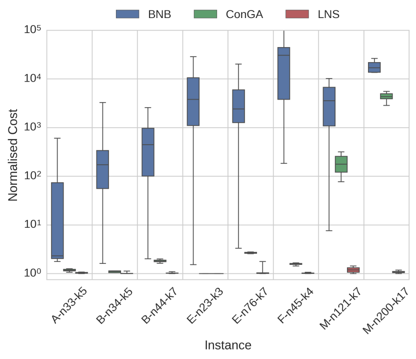
Fig. 2: Performance on BPP.

solution, while LNS and ConGA, whose sub-procedures are time-bound, fail to leverage the advantages of propagation. In the soft variants, on the other hand, the propagation is weaker, and the cost landscape is smoother, thus favouring both LNS and ConGA.

In the capacitated vehicle routing domain (Figs. 3a and 3b) B&B is consistently outperformed by ConGA, which is in turn outperformed by LNS. This is particularly true for the soft variant of the model. One thing to notice is that ConGA always finds at least one feasible solution, while B&B and LNS sometimes fail to do so in the hard variant of the model.



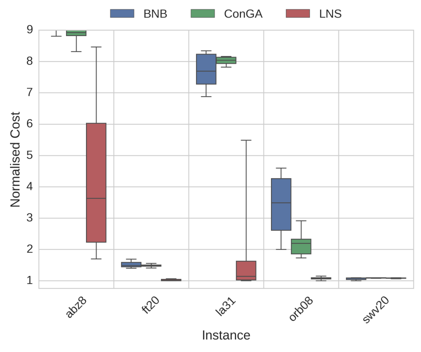
(a) Hard variant.



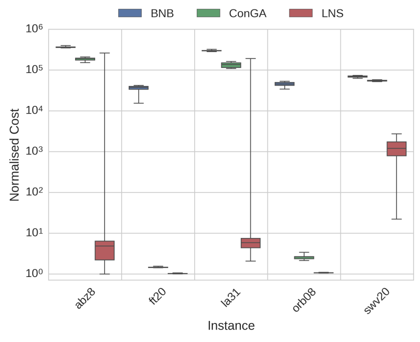
(b) Soft variant.

Fig. 3: Performance on CVRP.

In the job shop scheduling domain (Figs. 4a and 4b), we observe similar patterns both in the hard and the soft model variants, with B&B being outperformed by ConGA, and ConGA in turn being outperformed by LNS. On average, all of the approaches seem to achieve better performance in the hard variant of the model, with ConGA being outperformed by B&B in two cases.



(a) Hard variant.



(b) Soft variant.

Fig. 4: Performance on JSSP.

In the travelling salesperson problem domain (Fig. 5) we do not have a soft variant of the model (the model consists of a single *circuit* global constraint). The recurring pattern across all of the problem instances is that ConGA outperforms B&B but is outperformed (except in one case) by LNS.

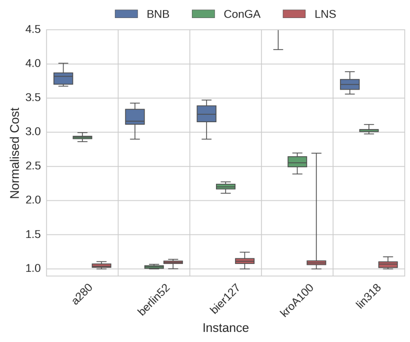


Fig. 5: Performance on TSP.

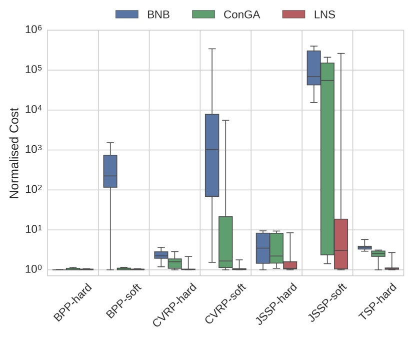


Fig. 6: Performance across problems.

3.4 Discussion

While there are small differences across the various problem domains, the results paint a reasonably consistent picture: ConGA almost consistently outperforms B&B, and LNS almost consistently outperforms ConGA. This is reflected in Figure 6 which shows the distribution of normalised costs across all of the instances for the various algorithms and model variants.

This overall trend has two exceptions. First, in problem domains that are relatively easy, e.g., the bin packing problem, B&B dominates the hard variants of the model, where it can successfully exploit the strength of constraint propagation and efficiently prune the search space. Secondly, on hard instances of some problem domains both B&B and LNS fail to find even one feasible solution, and therefore timeout. In these situations, ConGA succeeds more often in initialising the population and then proceeds reducing the cost. This happens because ConGA's initialisation process is time-bound, and can restart if it fails to find a feasible solution.

The superiority of ConGA over B&B is not surprising, as we are giving up completeness (guarantee to find an optimum if one exists, and to prove that the problem is unsolvable otherwise) in exchange for a more aggressive optimisation strategy. Regarding the consistently worse performance of ConGA with respect to LNS, we can point out at least three probable causes. The first, and probably most important, cause is the way computational resources are used. While LNS is a single-solution approach, in which the whole time budget can be invested into the improvement of a single incumbent solution, ConGA has to evolve a whole population of solutions. Therefore, only $\frac{1}{\mu}$ of the time budget can be spent on each *solution trajectory*. Fortunately, genetic algorithms are *embarrassingly parallel* [10], which means that this is likely a weakness of our prototypical single-thread implementation, rather than of the approach itself. Our intuition is that once parallelism is introduced in the framework, ConGA will outperform LNS, at least on highly concurrent environments such as clusters. Such intuition is supported by the results described in [22] regarding the performance speed-up that can be obtained when running several independent optimisation algorithms in parallel. A second likely cause for the reduced performance is poor parameter configuration. While, in principle, this could also be the case for LNS, the impact of various parameter settings on ConGA is not well understood yet, while the parameters used for LNS are sensible defaults that have proven to be reasonably safe in various contexts. Finally, except for the contribution of constraint propagation, it is fair to say that ConGA represents a rather simplistic genetic algorithm. The evolutionary computation community has studied, over the decades, several ways to design and advance genetic algorithms, e.g., mechanisms for preserving diversity, elitism techniques, better selection strategies, etc., whose inclusion into ConGA is still to be attempted. We expect in particular that diversity management mechanisms will be fundamental, as the *constrainedness* of our crossover and mutation operators, if anything, promotes convergence.

All in all, the preliminary results of our approach are encouraging, and clearly indicate the future directions of this work.

4 Conclusions & Future Work

Two common challenges arise when applying genetic algorithms to combinatorial optimisation problems: *i)* modelling and *ii)* constraint handling. Interestingly, these are also the main selling points of constraint programming, a constraint satisfaction paradigm based on the concept of propagation and domain filtering.

In this work, we proposed a general hybrid framework to combine the strengths of genetic algorithms and constraint programming. From genetic algorithms, we inherit the capability to explore large search spaces efficiently and identify high-quality solutions. From constraint programming, we inherit the modelling language of constraints and the constraint propagation techniques developed within the constraint programming community for more than 4 decades. In particular, one of the main strengths of our approach is the use of constraint propagation in our genetic operators to enforce properties such as offspring feasibility and quality.

The preliminary results obtained by our approach on a number of classical combinatorial optimisation problems demonstrate its general applicability, however there is much work to be done. Even if our method outperforms B&B, the standard search strategy in constraint programming, it is still not competitive with a general optimisation approach with comparable requirements: CP-based large neighbourhood search (LNS). We have discussed the probable causes of these results, which clearly indicate the next steps for this work.

First, we want to explore one of the most attractive features of genetic algorithms: parallelisability. This is of course motivated by the increasing availability of cheap multi-core hardware, and has the potential to give our approach a significant edge over less parallelisable approaches. Secondly, like many other metaheuristics, our algorithm is sensitive to parameter settings. This compels us to explore parameter tuning (or algorithm configuration) as a way of better understanding its strengths and improving its performance. A particularly appealing direction, in this regard, is feature-based tuning (FBT, [2]), i.e., configuring the parameters of the algorithm on-the-fly, based on the properties (features) of the problem instance being solved. FBT has already been used successfully to automatically configure problem-specific solvers. With our model-based approach we could take the technique to the next step: using *model* features, e.g., number and degree of variables, volume of the search space, number of constraints, etc., to configure the parameters of the algorithm *independently* from the problem domain. Thirdly, as we mentioned before, the genetic algorithm template upon which our approach is based is rather simple. In the future we plan to leverage the advanced techniques and operators developed within the evolutionary computation community, and to integrate them in our framework. All of the above are exciting research avenues that we intend to explore in the next stages of this project. From an engineering standpoint, we aim to generalise the constraint programming side of the framework, enabling the use of any constraint solver, not necessarily GECODE. To do this, we plan to exploit the API provided by the MINIZINC project, which offers a common modelling language and interface on top of a collection of different solvers.

Finally, while we have shown the applicability of our approach by solving some classical combinatorial optimisation problems, we want to assess its suitability to handle challenging real-world problems.

Bibliography

[1] Beasley, J.E.: Or-library: distributing test problems by electronic mail. Journal of the operational research society 41(11), 1069–1072 (1990)

[2] Bellio, R., Ceschia, S., Di Gaspero, L., Schaerf, A., Urli, T.: Feature-based tuning of simulated annealing applied to the curriculum-based course timetabling problem. Computers & Operations Research pp. 83–92 (2016)

[3] Davis, L.: Handbook of genetic algorithms (1991)

[4] Delorme, M., Iori, M., Martello, S.: Bin packing and cutting stock problems: Mathematical models and exact algorithms. European Journal of Operational Research 255(1), 1–20 (2016)

[5] Deris, S., Omatu, S., Ohta, H., Saad, P.: Incorporating constraint propagation in genetic algorithm for university timetable planning. Engineering Applications of Artificial Intelligence 12(3), 241–253 (1999)

[6] Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edn. (1989)

[7] Khichane, M., Albert, P., Solnon, C.: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, vol. 6140 (10 2010)

[8] Laporte, G., Musmanno, R., Vocaturo, F.: An adaptive large neighbourhood search heuristic for the capacitated arc-routing problem with stochastic demands. Transportation Science 44(1), 125–135 (2010)

[9] Michalewicz, Z., Dasgupta, D., Riche, R.G.L., Schoenauer, M.: Evolutionary algorithms for constrained engineering problems. Computers & Industrial Engineering 30(4), 851–870 (1996)

[10] Munawar, A., Wahib, M., Munetomo, M., Akama, K.: A survey: Genetic algorithms and the fast evolving world of parallel computing. In: High Performance Computing and Communications, 2008. HPCC'08. 10th IEEE International Conference on. pp. 897–902. IEEE (2008)

[11] Pisinger, D., Ropke, S.: Large neighborhood search. In: Handbook of metaheuristics, pp. 399–419. Springer (2010)

[12] Quimper, C.G., Rousseau, L.M.: A large neighbourhood search approach to the multi-activity shift scheduling problem. Journal of Heuristics 16(3), 373–392 (2010)

[13] Reinelt, G.: Tsplib - a traveling salesman problem library. ORSA journal on computing 3(4), 376–384 (1991)

[14] Ropke, S., Pisinger, D.: An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. Transportation science 40(4), 455–472 (2006)

[15] Rossi, F., Van Beek, P., Walsh, T.: Handbook of constraint programming. Elsevier (2006)

[16] Sastry, K., Goldberg, D.E., Kendall, G.: Genetic algorithms. In: Search methodologies, pp. 93–117. Springer (2014)

[17] Schulte, C.: Programming branchers. In: Schulte, C., Tack, G., Lagerkvist, M.Z. (eds.) Modeling and Programming with Gecode (2016), corresponds to Gecode 5.0.0

[18] Schulte, C., Tack, G., Lagerkvist, M.Z.: Modeling. In: Schulte, C., Tack, G., Lagerkvist, M.Z. (eds.) Modeling and Programming with Gecode (2016), corresponds to Gecode 5.0.0

[19] Shaw, P.: Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems, vol. 1520 (1998)

[20] Uchoa, E., Pecin, D., Pessoa, A., Poggi, M., Vidal, T., Subramanian, A.: New benchmark instances for the capacitated vehicle routing problem. European Journal of Operational Research 257(3), 845–858 (2017)

[21] Urli, T., Brotánková, J., Kilby, P., Van Hentenryck, P.: Intelligent habitat restoration under uncertainty. In: AAAI. pp. 3908–3914 (2016)

[22] Verhoeven, M.G.A., Aarts, E.H.L.: Parallel local search. Journal of Heuristics 1(1), 43–65 (1995)

[23] Vidal, T., Crainic, T.G., Gendreau, M., Prins, C.: A hybrid genetic algorithm with adaptive diversity management for a large class of vehicle routing problems with time-windows. Computers & operations research 40(1), 475–489 (2013)

[24] Wang, S., Chen, J., Wang, K.J.: Resource portfolio planning of make-to-stock products using a constraint programming-based genetic algorithm. Omega 35(2), 237–246 (2007)

[25] Wu, J., Shekh, S., Sergiienko, N.Y., Cazzolato, B.S., Ding, B., Neumann, F., Wagner, M.: Fast and effective optimisation of arrays of submerged wave energy converters. In: Proceedings of the 2016 on Genetic and Evolutionary Computation Conference. pp. 1045–1052. ACM (2016)