Benchmarking and Solving Dynamic Constrained Problems

Trung Thanh Nguyen, Xin Yao, Fellow, IEEE

Abstract—Many real-world dynamic optimisation problems have constraints, and in certain cases not only the objective function changes over time, but the constraints also change as well. However, in academic research there is not much research on continuous dynamic constrained optimization, and particularly there is little research on whether current numerical dynamic optimization algorithms would work well in dynamic constrained environments nor there is any numerical dynamic constrained benchmark problems. In this paper, we firstly investigate the characteristics that might make a dynamic constrained problems difficult to solve by existing dynamic optimization algorithms. We then introduce a set of numerical dynamic benchmark problems with these characteristics. To verify our hypothesis about the difficulty of these problems, we tested several canonical dynamic optimization algorithms on the proposed benchmarks. The test results confirm that dynamic constrained problems do have special characteristics that might not be solved effectively by some of the current dynamic optimization algorithms. Based on the analyses of the results, we propose a new algorithm to improve the performance of current dynamic optimization methods in solving numerical dynamic constrained problems. The test results show that the proposed algorithm achieves superior results compared to the tested existing dynamic optimization algorithms.

I. INTRODUCTION

Many real-world dynamic optimisation problems have constraints, and in certain cases not only the objective function changes over time, but the constraints also change as well. Some recent published examples are the ship scheduling problems [9], the adaptive farming strategies [8] and the aerodynamic/structural wing design problems [14].

Given the fact that dynamics in the constraints might also have consequences to the dynamics of the problems, it is then questioned that whether current continuous dynamic optimization strategies, which are designed and tested mainly on unconstrained dynamic problems, would still work in solving numerical dynamic constrained optimisation problems (DCOPs)? Unfortunately, this question has not been addressed extensively, because so far in academic research there is not much research on real-valued dynamic constrained optimization, and particularly there is little research on whether current dynamic optimization algorithms would work well in real-valued dynamic constrained environments nor there is any numerical dynamic constrained benchmark problems. The only numerical benchmark problem that has some properties related to dynamic constraints that we are aware of is the dynamic visibility test problems of Weicker and Weicker [16]. In this test problem, a mask of visibility is used to hide certain regions in the search space. This type of visibility mask, however, is not a dynamic constraint and does not have the property of a dynamic constraint.

In an attempt to answer the above question, in this paper we will investigate the characteristics of DCOPs that might cause difficulties to existing dynamic optimization strategies. Because there is no existing dynamic constrained benchmark available to verify our hypothesis, we will also introduce a new set of dynamic constrained benchmark problems to simulate the aforementioned characteristics. The test problems then are used to analyze the performance of two canonical dynamic optimisation algorithms, hyper-mutation GA [4] and random-immigrant GA [7].

Based on the result, we will propose a new algorithm named RepairGA to work with DCOPs. It uses the repair method from GENOCOP III [10] to track feasible areas and has a special method to maintain diversity.

The structure of the paper is organized as follows. Section 2 represents the characteristics that might make DCOPs difficult for existing dynamic optimization strategies to solve. Section 3 introduces the dynamic constrained benchmark that represents these characteristics. Section 4 evaluates the performance of existing dynamic optimisation algorithms and section 5 proposes a new approach to solve DCOPs: tracking feasible areas using constraint handling techniques. Finally, the discussion and conclusion are presented in section 6.

II. CHARACTERISTICS THAT MAKE DYNAMIC CONSTRAINED PROBLEMS HARD

Real-valued DCOPs have some special, but common characteristics that do not exist in unconstrained dynamic problems. Some of these characteristics might make the behaviours of DCOPs different from their unconstrained counterparts, hence cause some difficulties for current dynamic optimisation algorithms, which are designed for unconstrained problems, to solve. The characteristics, and the difficulties they bring to current methods, are as follows:

1) The presence of infeasible areas might make maintaining/introducing diversity approaches less effective: In solving unconstrained dynamic problems, a common approach is to either maintain diversity during the search or increase diversity whenever a change is detected. The diversified individuals are used for two purposes: (1) to detect new appearing optima and (2) to find the new place of the moving optima. In DCOPs, however, just maintaining/introducing diversity might not be enough because many diversified individuals might be infeasible and hence do not contribute to the two purposes mentioned above.
(2) The switching of global optima between disconnected feasible regions might make methods that use penalty functions less effective: The simplest way to apply current dynamic optimization methods to solving constrained dynamic problems, without using any specialized constraint handling techniques, is to use penalty functions. However, the use of penalty functions might not be effective in problems where there exists multiple disconnected feasible regions. In these problems, due to the dynamic of the objective function and/or the constraint functions the global optimum might switch from one disconnected region to another. In this case in order to track the moving optimum from one region to another, it is necessary to have a path going through the infeasible areas that separate the disconnected regions. This path might not be available if we use penalty functions because penalties make it unlikely that infeasible individuals are accepted.

(3) The moving constraints might make tracking the previous global optimum less effective: Another common approach used in unconstrained dynamic optimization is to track the moving global optimum. Whenever a change is detected, either a high rate of mutation is introduced around the optimum, or a sub-population is used to monitor the optimum. It is hoped that these two methods would help to track the moving optimum if it moves gradually in small enough distances. In DCOPs, however, this approach might not be as effective as it is in the unconstrained case. In certain situations, the whole area around the previous optimum might suddenly become infeasible, and the algorithm would have no clue of how to get out of this infeasible area to track the new optimum. In some other situations, the moving constraints might expose better regions with a better optimum without changing the area where the previous optimum is in. Because the value of the previous global optimum does not change, algorithms focusing on tracking the current optimum would not be noticed about the change and consequently would not be able to find the new optimum. In summary, tracking the previous optimum does not guarantee that we can find the best result in DCOPs.

III. A DYNAMIC CONSTRAINED BENCHMARK SET

As mentioned in section 1, because there is no existing numerical dynamic constrained benchmark, it is difficult to verify whether a dynamic optimization algorithm would work well in a DCOP. To assist researchers in designing their algorithms, we proposed a set of dynamic constrained benchmark problems. The set contains both unimodal and multi-modal, scalable problems. Details of the problems, as well as the source code, are set out in [13]. All problems are based on static benchmark problems using the idea belows:

Given a static function $f_P(x)$ with a set of parameter $P = \{p_1, \ldots, p_k\}$, we can always generalise $f_P(x)$ to its dynamic version $f_{\tau}(x,t)$ by replacing each static parameter $p_i \in P$ with a time-dependent expression $p_i(t)$. The dynamic of the time-dependent problem then depends on how $p_i(t)$ varies over time. Details of the concept and a mathematical framework for the idea is set out in [12].

![Fig. 1. One instance of the G24 (G24.4) landscape at different changes. We can see that both the dynamic objective function and the constraints change over time. The size and shape of the feasible areas, and the number of disconnected regions also change accordingly.](image)

Using this idea, in this paper we introduce five instances of a dynamic problem named G24, chosen from our set of benchmarks in [13]. This problem is formulated based on a static function proposed in [6]. The description of the problem is set out in eq. 1, the properties of each instances are set out in table I, the illustration of one instance (G24.4) is described in figure 1 and the dynamic parameters of each instance is described in table II. The five instances have all characteristics of dynamic constrained problems, as metioned in section 2 (see table I).

\[
f(x) = - (X_1(x_1, t) + X_2(x_2, t)) \tag{1}
\]

subject to:

\[
g_1(x,t) = -2Y_1(x_1, t)^4 + 8Y_1(x_1, t)^3 - 8Y_1(x_1, t)^2 + Y_2(x_2, t) - 2 \leq 0
\]

\[
g_2(x,t) = -4Y_1(x_1, t)^4 + 32Y_1(x_1, t)^3 - 88Y_1(x_1, t)^2 + 96Y_1(x_1, t) + Y_2(x_2, t) - 36 \leq 0
\]

where

\[
X_1(x, t) = p_i(t)(x + q_i(t))
\]

\[
Y_1(x, t) = r_i(t)(x + s_i(t))
\]

\[
0 \leq x_1 \leq 3; 0 \leq x_2 \leq 4
\]

$p_i(t), q_i(t), r_i(t) \quad \text{and} \quad s_i(t) \quad (i = 1, 2)$ are the time-dependent parameters that determine how the function would change over time. They are described in table II.

IV. EVALUATING HYPER-MUTATION GA AND RANDOM IMMIGRANT GA ON THE BENCHMARK

To evaluate our hypotheses (in section 2) that some characteristics in DCOPs might prevent current dynamic optimization strategies from getting the best results, we test the benchmark problems with two canonical dynamic optimization algorithms: hyper-mutation GA (hyperM) [4] and random-immigrant GA (RIGA)[7]. HyperM represents
TABLE I
PROPERTIES OF FIVE TESTED PROBLEMS.

<table>
<thead>
<tr>
<th>Problem</th>
<th>ObjFunc</th>
<th>Constr</th>
<th>DFR</th>
<th>SwGO</th>
<th>GIB</th>
<th>NAO</th>
</tr>
</thead>
<tbody>
<tr>
<td>G24_0</td>
<td>Cyclic</td>
<td>No</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>G24_1</td>
<td>Cyclic</td>
<td>Fixed</td>
<td>2</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>G24_2</td>
<td>Cyclic</td>
<td>Fixed</td>
<td>2</td>
<td>Yes</td>
<td>Y&amp;No</td>
<td>No</td>
</tr>
<tr>
<td>G24_3</td>
<td>Fixed</td>
<td>Linear</td>
<td>1-3</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>G24_4</td>
<td>Cyclic</td>
<td>Linear</td>
<td>1-3</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>G24_5</td>
<td>Cyclic</td>
<td>Linear</td>
<td>1-3</td>
<td>Yes</td>
<td>Y&amp;No</td>
<td>No</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DFR</th>
<th>SwGO</th>
<th>NAO</th>
<th>GIB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of disconnected feasible regions</td>
<td>Global opt. switches among disconnected regions</td>
<td>Newly appearing optima without changing existing optima</td>
<td>Global optimum is in the boundary of feasible areas</td>
</tr>
</tbody>
</table>

TABLE II
PARAMETER SETTINGS FOR FIVE TESTED PROBLEMS.

<table>
<thead>
<tr>
<th>Prob</th>
<th>Parameter settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>G24_0</td>
<td>( p_1(t) = \sin \left( \frac{k \pi t}{2} \right) ; p_2(t) = 1; q_{1,2}(t) = 0 )</td>
</tr>
</tbody>
</table>
| G24_1 | \( p_2(t) = r_1(t) = 1; q_i(t) = s_i(t) = 0; i = 1,2 \)  
\( p_1(t) = \sin \left( \frac{k \pi t}{2} \right) ; 0 < k \leq 2 \) |
| G24_2 | \( \begin{cases} 
  p_1(t) = \sin \left( \frac{k \pi t}{2} \right) & \text{if } \left( t \mod 2 \right) = 0 \\
  p_2(t) = \sin \left( \frac{k \pi t}{2} \right) & \text{if } \left( t \mod 2 \right) \neq 0 
\end{cases} \)  
\( q_i(t) = s_i(t) = 0; r_1(t) = 1; i = 1,2 \) |
| G24_3 | \( s_2(t) = 2 + \frac{k^2 \max - k^2 \min}{S} \) |
| G24_4 | \( s_2(t) = t \frac{k^2 \max - k^2 \min}{S} \) |
| G24_5 | \( \begin{cases} 
  p_1(t) = \sin \left( \frac{k \pi t}{2} \right) & \text{if } \left( t \mod 2 \right) = 0 \\
  p_2(t) = \sin \left( \frac{k \pi t}{2} \right) & \text{if } \left( t \mod 2 \right) \neq 0 
\end{cases} \)  
\( q_i(t) = s_i(t) = 0; r_1(t) = 1; i = 1,2 \) |

<table>
<thead>
<tr>
<th>Prob</th>
<th>Parameter settings</th>
</tr>
</thead>
</table>
| \( k \) | Determines the severity of function changes.  
\( k = 1 \sim \text{large}; k = 0.5 \sim \text{medium}; k = 0.25 \sim \text{small} \) |
| \( S \) | Determines the severity of constraint changes  
\( S = 10 \sim \text{large}; S = 20 \sim \text{medium}; S = 50 \sim \text{small} \) |

To create a fair testing environment, the parameters of all tested algorithms are set to similar values if possible. The strategy parameters of RIGA and hyperM are set to the values used in the original papers ([7] and [5]). The values of all parameter settings are listed in table III. Due to the limited space of this paper, we will only present the tested results in three DCOPs G24_1, G24_3 and G24_4. Because G24_2 and G24_5 have the same property as G24_1 and G24_4, respectively, it is expected that there is no significant difference between the performance of algorithms in these two sets of problems. We also test the dynamic unconstrained version of G24 (G24_0). Please note that the actual change severity in G24_0 is larger than the constrained versions (Due to the lack of constraints, in G24_0 the moving optimum travels a larger distance. As a result, the severity level is larger).

A. Test settings

B. Performance measures

To measure the performance of the algorithms, in this paper we use two measures: the first one is the performance plots, based on the average best so far solution at each generation over many runs of the same problem [1]. The second measure is the offline error proposed in [3]. Offline error is measured as the average over, at every evaluations, the error of the best solution found since the last change of the environment. This measure is always greater than or equal to zero and would be zero for a perfect performance. In all experiments presented in this paper we use the performance of standard GA as the baseline to evaluate the performance of other algorithms. This is because all other tested algorithms are developed from GA and the difference between them and GA is that they have additional mechanisms to handle dynamic (e.g. increasing the mutation rate whenever a change is detected). Comparing the tested algorithms with GA in different problems would help us evaluate how well those dynamic-handling mechanisms work in different situations.
<table>
<thead>
<tr>
<th>Parameter Settings for Testing Algorithms on the Benchmark Functions</th>
<th>All algorithms (with exceptions shown below)</th>
<th>HyperM</th>
<th>dHyperC</th>
<th>RIGA</th>
<th>RepairGA</th>
<th>GENOCOP</th>
<th>Problem settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pop size</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>Elitism</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Selection method</td>
<td>Roulette wheel</td>
<td>Uniform, $P = 0.001$ [5]</td>
<td>Uniform, $P = 0.5$</td>
<td>Uniform, $P = 0.3$ [7]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mutation method</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Crossover method</td>
<td>Arithmetic, $P = 0.8$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Search population size</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference pop size</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Selection method</td>
<td>Non-linear ranking as in [10]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of runs</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Change frequency</td>
<td>1000 evaluations</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function severity $k$</td>
<td>0.5 (medium)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constraint severity $S$</td>
<td>20 (medium)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C. Overall results

Figure 2 and table IV presents the performance plots and the offline errors of GA vs HyperM vs RIGA. The result reveals some interesting observations. First, HyperM and RIGA perform less effective in cases there are infeasible areas in the search space, i.e. in cases the dynamic problems have constraints (G24_1 and G24_4). For example, although both HyperM and RIGA are 1.57 and 1.69 times better than GA in the dynamic unconstrained case (G24_0), in cases where there are constraints (G24_1 and G24_4) RIGA is only 1.42 and 1.49 times better than GA and HyperM even performs worse than GA (equal to only 0.88 and 0.38 times the performance of GA). Because the only difference between G24_0 and G24_1/G24_4 are the constraints, these constraints might be the reason why hyperM and RIGA become less effective in solving G24_1 and G24_4. This partially verifies our first hypothesis (in section 2), at least in the tested problems.

Second, all algorithms perform worse in tested problems that have disconnected feasible regions. For example the performance of GA/HyperM/RIGA in G24_0 (where there is one feasible region) is 0.45, 0.27 and 0.29, respectively, while their performance in G24_1 (where there exists two disconnected feasible regions) are 0.8, 0.65 and 0.57 even that the change severity in G24_0 is higher. Because the only difference between G24_0 and G24_1 is that G24_1 has an infeasible area that separates the search space into two disconnected regions with the global optimum switches from one to the other, there might be the possibility that the worsen performance of algorithms in solving G24_1 might be due

Fig. 2. Plots of best solutions so far since the last change found by GA, hyper-mutation GA (hyperM) and random-immigrant GA (RIGA) in G24_0 (top), G24_3 (middle) and G24_4 (bottom). Although hyperM and RIGA outperform GA in the unconstrained case (G24_0) and fixed constraint case (G24_1, not shown), they do not perform so well in cases the constraints are dynamic. Especially, HyperM performs significantly worse than GA in G24_3 and G24_4. It is also unable to detect changes in G24_3.
TABLE IV
THE OFFLINE ERROR OF GA, HYPERM AND RIGA.

<table>
<thead>
<tr>
<th>Error</th>
<th>StDev</th>
<th>vsGA**</th>
<th>Error</th>
<th>StDev</th>
<th>vsGA**</th>
</tr>
</thead>
<tbody>
<tr>
<td>G24.0 (dynF+noC)</td>
<td></td>
<td></td>
<td>G24.3 (fixF+dynC)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GA</td>
<td>0.4488</td>
<td>0.049</td>
<td>1.00</td>
<td>0.9760</td>
<td>0.127</td>
</tr>
<tr>
<td>RIGA</td>
<td>0.2854</td>
<td>0.043</td>
<td>1.57</td>
<td>0.6664</td>
<td>0.063</td>
</tr>
<tr>
<td>hyperM</td>
<td>0.2660</td>
<td>0.012</td>
<td>1.69*</td>
<td>1.1079</td>
<td>0.482</td>
</tr>
<tr>
<td>G24.1 (dynF+fixC)</td>
<td></td>
<td></td>
<td>G24.4 (dynF+dynC)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GA</td>
<td>0.8117</td>
<td>0.077</td>
<td>1.00</td>
<td>0.8842</td>
<td>0.081</td>
</tr>
<tr>
<td>RIGA</td>
<td>0.5734</td>
<td>0.076</td>
<td>1.42*</td>
<td>0.5937</td>
<td>0.054</td>
</tr>
<tr>
<td>hyperM</td>
<td>0.6472</td>
<td>0.271</td>
<td>1.25</td>
<td>2.3370</td>
<td>1.942</td>
</tr>
</tbody>
</table>

* better results
** indicate how many times the tested algorithm is better than GA

V. TRACKING MOVING FEASIBLE AREAS

A. Can tracking feasible areas help in solving dynamic constrained problems?

The result in section 4 shows us that although unconstrained dynamic optimization strategies as introducing/maintaining diversity and tracking previous global optima still helps in the constrained cases, they are not effective enough to achieve the best results, at least for the tested problems. As a result, it is necessary to combine them with new strategies to better exploit the characteristics of DCOPs, as well as to handle the constraints more effectively.

Also from section 4, as mentioned earlier we have observed an interesting situation in G24_4 when the problem becomes easier to solve if the algorithm is able to track the moving feasible areas. This observation suggests us that in order to improve the performance of algorithms in solving DCOPs, we can try to improve the ability to track the moving feasible areas.

One way to track the moving feasible areas might be to always keep some members of the population feasible. These members will be used as pivot points for us to know where the feasible areas currently are. At every generation, we will have to update these members to make sure that they are always feasible. Then during the search, these members can be used as reference to repair any infeasible individuals. By converting infeasible solutions to feasible ones like this, we actually attract individuals to the feasible areas, or in other words track the movement of the feasible areas. This way, the diversified individuals generated for maintaining/introducing diversity can be used more effectively. In addition, because after a change the global optimum either move along with the feasible areas, or appear in a new feasible area, tracking feasible areas increases the possibility that we can track the actual global optimum.

B. The RepairGA algorithm

To implement the above idea, we need to firstly find a way to update the group of feasible solutions whenever a change happen and secondly we need an effective method to repair infeasible solutions. There are several available repair methods in the literature. In this paper we use the repair operator proposed in GENOCOP III [10] because it is most suitable for our purpose: it supports a reference population containing all feasible solutions and a repair method to convert infeasible solutions to feasible ones using that reference population. For a points s that is infeasible, the repair method tries to "repair" it using another point r from the reference population by generating random points from a segment between s and r until a feasible point is found.

To be able to track feasible areas, the repair operator needs to be modified so that the reference population is always updated during the search. There are two types of updates. The first type of update is carried out at every generation to make sure that every reference individuals are feasible. In this type of update every reference individuals that are found
infeasible will be repaired to be feasible again using other feasible solutions.

The second type of update is triggered after a certain period to maintain diversity and to avoid the possibility that the whole population has been trapped in one single disconnected feasible region. When being triggered each individual of the reference population is mutated and crossovered using uniform mutation and arithmetic crossover with a certain probability. If the offspring solutions are feasible they will replace their parents. In this experiment the update period is set to 100 evaluation cycles. One evaluation cycle is counted when any individual is re-evaluated or when a pair of crossover/mutation operators is made on any individual.

Then the repair method needs to be integrated into GA so that we can compare its performance to hyperM/RIGA in a fair environment. All these tasks have been implemented in a new algorithm called RepairGA. Details of the algorithm is set out in algorithm 2 and its Repair routine in algorithm 3.

It is also worth to notice that different from GA, HyperM and RIGA, RepairGA does not use the normal roulette wheel selection method. Instead it uses the non-linear ranking selection method specified in [10]. The reason for us to use this selection method is to be able to compare the performance of RepairGA with GENOCOP III. We have also tested RepairGA using roulette wheel selection and normal ranking selection and observed no significant difference in the performance of RepairGA compared to the case of using non-linear ranking selection. Using non-linear ranking selection, the selection wheel for a population with size \( N \) (sorted in descending order) is calculated using algorithm 1.

**Algorithm 1**

```plaintext```
1) \( f = 0.1; q = f / \left( 1 - (1 - f)^N \right) \)
2) For \( i = 1 : N \)
   a) \( RW[i] = q^i (1 - q)^{i-1} \)
3) For \( i = 1 : N - 1 \)
   a) \( RW[i + 1] = RW[i + 1] + RW[i] \)
```

**C. Experimental results**

The performance of RepairGA was compared with HyperM and RIGA in the tested problems (figure 3 and table V). The total number of evaluation per run is 10,000 (10 changes). We are interested in dynamic constrained problems, which are the problems where GA, HyperM and RIGA perform less effective. The results show that RepairGA significantly outperforms both hyperM and RIGA in all dynamic constrained problems (by a factor of between 75.13 in G24_3 and 8.54 in G24_4). RepairGA also finds all global optima and recovers from a drop after a change much quicker than hyperM and RIGA. It means that RepairGA might be able to track the moving feasible areas (in G24_3 and G24_4) and go through the infeasible areas between disconnected regions (in G24_1 and G24_4) much better than hyperM and RIGA.

Fig. 3. Plots of best solutions found so far since last change by HyperM, RIGA and RepairGA. It can be seen that RepairGA outperforms both HyperM and RepairGA in all dynamic constrained problems. It has smaller performance drop after each change, recover after each change faster and also found the global optima in all cases.
Algorithm 2 RepairGA

1) Initialise:
   a) Initialise n individuals in the search population S
   b) Initialise n individuals in the reference population R
      i) Randomly generate points until a feasible is found
      ii) Add the feasible point to R
      iii) Repeat step 1(b) until n individuals are found
2) Search: For i = 1 : n
   a) Crossover: Using nonlinear ranking selection to choose a pair of parents from S
      i) Crossover an offspring s from the chosen parents
      ii) Evaluate s
      iii) Repair s
         A) Randomly pick an individual r ∈ R
         B) Create a feasible individual s1 from s and r by calling s1 = Repair(s, r)
         C) If f(s1) better than f(r) then r = s1
         D) If U(0, 1) < PRepair: s = s1
   b) Mutation: Using nonlinear ranking selection to choose a parent from S
      i) Mutate an offspring s from the chosen parent
      ii) Evaluate s
      iii) Repair s
         A) Randomly pick an individual r ∈ R
         B) Create a feasible individual s1 from s and r by calling s1 = Repair(s, r)
         C) If f(s1) better than f(r) then r = s1
         D) If U(0, 1) < PMutation: s = s1
      iv) Using nonlinear ranking selection to replace one of the worst individuals in S by s
3) Update the reference population R:
   a) Evaluate all individuals in R
   b) For each r ∈ R: r is infeasible
      i) Find a feasible point x to repair r
         A) If ∃ s ∈ S: x is feasible, x = s
         B) Otherwise, if ∃ q ∈ R: q is feasible, x = q
         C) Otherwise, reinitialize x using it is feasible
      ii) Repair r: r = Repair(r, x)
4) Evolve the reference population after each 100 evaluation cycles: For i = 1 : n
   a) Mutation: Using nonlinear ranking selection to choose a parent x from R
      i) Mutate an offspring s from x
      ii) If s is feasible, replace x = s
   b) Crossover: Using nonlinear ranking selection to choose a pair of parent from R
      i) Crossover an offspring s from the chosen parents
      ii) If s is feasible, replace the worse parents with s
5) Return to step 2

Algorithm 3 routine Repair(Indiv s, Indiv r)
1) Generate individual x in the segment between s and r
   a) a = U(0, 1)
   b) x = a.s + (1 - a).r
2) While x is infeasible, back to step 1
3) If a feasible x is not found after 100 trials, x = r
4) If f(x) better than f(r) then x = r
5) Return the individual x

**TABLE V**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Error</th>
<th>StDev</th>
<th>vsGA**</th>
</tr>
</thead>
<tbody>
<tr>
<td>G24_0 (dynF=noC)</td>
<td>0.2854</td>
<td>0.043</td>
<td>1.57</td>
</tr>
<tr>
<td>RIGA</td>
<td>0.6664</td>
<td>0.063</td>
<td>1.46</td>
</tr>
<tr>
<td>HyperM</td>
<td>0.2660</td>
<td>0.012</td>
<td>1.69</td>
</tr>
<tr>
<td>RepairGA</td>
<td>1.1079</td>
<td>0.482</td>
<td>0.88</td>
</tr>
<tr>
<td>G24_4 (dyF+fixC)</td>
<td>0.0148</td>
<td>0.002</td>
<td>1.77*</td>
</tr>
<tr>
<td>RIGA</td>
<td>0.6472</td>
<td>0.271</td>
<td>1.25</td>
</tr>
<tr>
<td>HyperM</td>
<td>2.3370</td>
<td>1.942</td>
<td>0.38</td>
</tr>
<tr>
<td>RepairGA</td>
<td>0.0448</td>
<td>0.009</td>
<td>18.13*</td>
</tr>
</tbody>
</table>

* better results
** indicate how many times the tested algorithm is better than GA

The reason for the good performance of RepairGA might be because it is able to deal with all three issues of dynamic constrained algorithms, as listed out in section 2. For the first issue: inefficiency of infeasible diversified solutions, RepairGA is able to convert those infeasible individuals into feasible ones, hence concentrates the diversification to the feasible areas to speed up the search process.

For the second issue: switching of optima between disconnected feasible regions requires paths through infeasible areas, by generating random points from a segment between a feasible point and an infeasible point, RepairGA is also able to establish a path through the infeasible areas, making way for the algorithm to travel among disconnected regions.

For the third issue: the moving constraints might make tracking the previous global optimum ineffective, because RepairGA focuses on tracking the feasible areas rather than the previous global optimum, it would not be deceived and would be able to find the actual global optimum.

We also compare the performance of RepairGA with HyperM and RIGA in the dynamic unconstrained problem (G24_0). In this problem the behaviour of RepairGA is similar to that of HyperM and RIGA, and its performance is also similar to that of HyperM and RIGA, although slightly better. The result in G24_0 shows that although RepairGA is designed to solve dynamic constrained problems, its mechanism of maintaining diversity and tracking changes is sufficient to work in unconstrained problems as well as (or even better than) HyperM and RIGA.

To investigate the trade-off of keeping the reference population always feasible, we also compare the performance
of RepairGA with GENOCOP III. Both GENOCOP III and RepairGA use the same repair operator. However, RepairGA is different from GENOCOP III in the fact that RepairGA always try to maintain diversity in the feasible search space by continuously updating the reference population to make sure that all members of this sub-population are feasible. In addition because RepairGA was extended from GA, it uses only two operators: mutation and crossover while GENOCOP III combines ten different operators.

The result is presented in table VI. It shows that although the additional tasks of maintaining diversity and feasibility make RepairGA perform less effective than GENOCOP III in the unconstrained problem G24_0 (by a factor of 12.4), they help RepairGA outperforms GENOCOP III in all dynamic constrained problems (by a factor of between 1.84 and 3.5).

**TABLE VI**

<table>
<thead>
<tr>
<th>Error</th>
<th>StdDev</th>
<th>vsGA**</th>
<th>Error</th>
<th>StdDev</th>
<th>vsGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>G24_0 (dynF+noC)</td>
<td>G24_3 (fixF+dynC)</td>
<td>RepairGA</td>
<td>0.2531</td>
<td>0.026</td>
<td>1.77</td>
</tr>
<tr>
<td>GENOCOP</td>
<td>0.0204</td>
<td>0.015</td>
<td>21.59*</td>
<td>0.0273</td>
<td>0.007</td>
</tr>
<tr>
<td>G24_1 (dynF+fixC)</td>
<td>G24_4 (fixF+dynC)</td>
<td>RepairGA</td>
<td>0.0448</td>
<td>0.009</td>
<td>18.13*</td>
</tr>
<tr>
<td>GENOCOP</td>
<td>0.1020</td>
<td>0.025</td>
<td>7.96</td>
<td>0.2406</td>
<td>0.044</td>
</tr>
</tbody>
</table>

* better results
** indicate how many times the tested algorithm is better than GA

VI. CONCLUSIONS

In this paper we have firstly investigated three common characteristics of DCOPs that might cause difficulties to existing dynamic optimization strategies, which have been designed and tested mainly on unconstrained DOPs.

Then we introduced a new set of dynamic constrained benchmarks to represent these characteristics, and tested some canonical dynamic optimization strategies on the benchmarks.

The test results show that although such strategies as introducing/maintaining diversity and tracking previous global optima work well in the unconstrained case, in the tested DCOPs they are not as effective.

Based on the analyses of the results, we have proposed a new algorithm (RepairGA), which focuses on tracking the moving feasible areas and repairing infeasible solutions. It is shown that RepairGA has been able to overcome all three issues of existing dynamic optimization strategies and have achieved superior results to existing dynamic optimization algorithms that have been tested.

From the results, we can see that in addition to existing strategies for unconstrained optimization, to work well with numerical DCOPs such new strategies as making use of infeasible solutions (as in [15]), repairing infeasible solutions and tracking feasible areas (as in RepairGA) might need to be taken into account.

ACKNOWLEDGMENT

The authors would like to thank P. Rohlfshagen, T. Ray, L. Xing, Y. Jin and S. Yang for their fruitful discussions. This work is partially supported by an EPSRC grant (EP/E058884/1) on "Evolutionary Algorithms for Dynamic Optimisation Problems: Design, Analysis & Applications", an UK ORS Award and a studentship from School of Computer Science, University of Birmingham. The programs used in this paper are developed from the source code provided by Ken Williams [17] and Zbigniew Michalewicz [11].

REFERENCES


2009 IEEE Congress on Evolutionary Computation (CEC 2009) 697