Static vs. Dynamic Populations in Genetic Algorithms for Coloring a Dynamic Graph

Cara Monical[1]
Centre College
600 W Walnut St
Danville, KY 40422
cmonica2@illinois.edu

Forrest Stonedahl[2]
Centre College
600 W Walnut St
Danville, KY 40422
forrest.stonedahl@centre.edu

ABSTRACT

We studied the performance of genetic algorithms for coloring dynamic graphs under a variety of experimental conditions, focusing on the relationship between the dynamics of the graph and that of the algorithm. Graph coloring is a well-studied NP-hard problem, while dynamic graphs are a natural way to model a diverse range of dynamic systems. Dynamic graph coloring can be applied to online scheduling in a changing environment, such as the online scheduling of conflicting tasks. As genetic algorithms (GAs) have been effective for graph coloring and are adaptable to dynamic environments, they are a promising choice for this problem. Thus, we compared the performance of three algorithms: a GA that maintained a single population adapting to the dynamic graph (DGA), a GA that restarted with a fresh population for the static graph of each time-step (SGA), and DSATUR, a well-known heuristic graph coloring algorithm re-applied at each time-step. We examined the relative performance of these algorithms for dynamic graphs of different sizes, edge densities, structures, and change rates, using different amounts of evolution between time-steps. Overall, the DGA consistently outperformed the SGA, being particularly dominant at low change rates, and under certain conditions was able to outperform DSATUR.

Categories and Subject Descriptors

I.28 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search; G.2.2 [**Discrete Mathematics**]: Graph Theory—*graph algorithms*

General Terms

Algorithms

Keywords

genetic algorithms, combinatorial optimization, dynamical optimization, dynamic graph, graph coloring

GECCO'14, July 12–16, 2014, Vancouver, BC, Canada.

ACM 978-1-4503-2662-9/14/07 ...$15.00.
http://dx.doi.org/10.1145/2576768.2598233.

1. INTRODUCTION AND RELATED WORK

The graph coloring problem is the well-known problem of assigning as few colors as possible to the vertices of a graph such that no two adjacent vertices are given the same color and is commonly used for scheduling applications [15, 13, 5]. Many real-world problems, however, are better modeled by a dynamic graph that can reflect a changing environment [8]. In the case of scheduling applications, this would allow for the online assignment of resources when only the current tasks are known, such as registers for the currently known values in program execution, frequencies for the currently connected devices in a mobile ad-hoc network, or batches for the currently tasked jobs in a job management system.

In the static model, a graph G has a set of vertices, V, and a set of edges, E. A dynamic graph adds the dimension of time, and so a dynamic graph varies either V (*vertex-dynamic or node-dynamic*) or E (*edge-dynamic*) or both [8]. We consider a discrete dynamic graph that is a succession of (related) static graphs; a *time step* is the time between the changes of the dynamic graph and G_k denotes the static graph at time k. The simplest extension of the static graph coloring problem to the dynamic graph coloring problem is to find a minimal coloring of the vertices of G at each time step. Because the static graph coloring decision problem is NP-complete [7] and the optimization problem is NP-hard, the dynamic graph coloring problem must be NP-hard also. In this work, we only consider vertex-dynamic graphs where edges are added and removed only when one of their endpoints is added or removed.

Two considerations for dynamic graph coloring problems are (1) whether the complete evolution (an offline problem) or only the current state (an online problem) of the dynamic graph is known during execution, and (2) whether vertices are allowed to change color as the graph changes. If a vertex color is fixed for the lifetime of the vertex, the offline dynamic graph coloring problem reduces to the static graph coloring problem on the graph constructed by taking the union of V and E over all time steps. Thus, this case is relatively uninteresting. While Lovász et al. [12] consider an online graph coloring problem where vertices' colors are fixed after assignment, we will consider a relaxed variant of the online dynamic graph coloring problem that allows color re-assignment at each time step.

The goal for our dynamic graph coloring problem is to minimize the cumulative (or equivalently, average) number

[1]Now at: University of Illinois Urbana-Champaign, Math Department, 1409 W. Green Street, Urbana, IL 61801
[2]Affiliated with Augustana College as of August 2014

of colors used to (properly) color the graph over time. Even if the color of a vertex can change between time steps, the coloring at time t is still a natural starting point for the coloring at time $t+1$. Therefore, we could also consider *color turnover*, or how many vertices change colors between time steps. Minimizing the number of colors and minimizing the amount of color turnover are potentially competing goals (as preventing the recoloring of a vertex could prevent finding an optimal coloring).

There may be practical situations where the cost of recoloring a vertex is greater than the cost of a slightly suboptimal coloring (such as in register allocation where values would need to be moved); however, in this work, we allow vertices to change colors freely and do not consider color turnover. In contrast, an ant-colony-inspired approach is given in [14], where vertices are only recolored when a conflict is created or when the degree saturation (the number of colors in the neighbors of a vertex) can be lowered. This approach will tend to have a lower rate of color turnover.

Genetic algorithms (GAs) [10] have frequently been used on the static graph coloring problem with reasonable success [3]. Additionally, in dealing with a dynamic environment, GAs are a natural choice because they mimic natural evolution, which occurs in a changing environment. However, one challenge with adapting GAs to dynamic problems is updating individuals in the population so they correspond to valid solutions of the changed problem [11]. While the most success in static graph coloring using genetic algorithms has occurred by hybridizing partition-based genetic algorithms with local search [6, 5], these algorithms present several challenges in adapting to dynamic graph coloring.

These algorithms commonly start by trying to find a k_0-coloring and when that is successful, search for a k_0-1 coloring and so on, until they find a suitable solution or reach the limits of available computational resources. This approach is not well-suited to a highly dynamic system where the goal is to find an optimal proper coloring at each time step and there may be a limited amount of evolution between changes in the graph. Furthermore, with edge-dynamic graphs, a change in the graph could break the independent sets of the individuals in the population, requiring costly rearrangement to restore individuals corresponding to a valid coloring. The genetic algorithm would need to actively detect changes in the graph (rather than just adapting to them) in order to update the population accordingly. Finally, even with just vertex-dynamic graphs, a small change in the graph could cause significant shifts in the vertices that should be colored the same color as new vertices will create new interactions between existing vertices.

The two other main classes of GAs for graph coloring are integer-based algorithms that encode and evolve colorings of the graphs directly and permutation-based algorithms that encode and evolve permutations of the graph's vertices (genotype), which are then decoded into colorings (phenotype) using the standard greedy coloring algorithm [9]. In integer-based algorithms the colorings are evolved directly and so a small change in the graph could require significant changes in the individuals to restore a proper coloring. However, with permutation-based algorithms, the changes to the graph are much more easily transferred to the individuals: vertices that have been removed from or added to the graph merely need to be removed from or added to the ordering encoded by each individual. Additionally, as the graph changes, it is likely that high fitness individuals will retain their high fitness—key vertices for early coloring in the graph at time t are likely still key in the changed graph at $t+1$, despite (possibly large) changes in the phenotypic coloring. Furthermore, permutation-based GAs do not require an active awareness of the algorithm to changes in the graph as the decoder will detect changes in the graph when using an individual to produce a coloring. Thus, we have focused exclusively on permutation-based GAs.

There are two genres of problems, traveling salesman-like problems and scheduling-like problems, that are typically tackled with permutation-based GAs, each with its own set of effective crossover and mutation operators [17]. In the former genre, preserving adjacency within an individual is important, while in the latter, preserving relative order of the vertices is key. Graph coloring clearly falls in the second class of problems, as it is frequently used for scheduling.

To our knowledge, we are the first to employ GAs for the dynamic graph coloring problem. As such, we focus on exploring the parameters of both the problem (Section 2.1) and the algorithm (Section 2.2) that affect the performance of a permutation-based GA. In Section 3, we discuss the effects on these parameters, including the edge density and graph structure of the graph (3.1), the rate at which the graph is changing (3.2), and the amount of evolution that can be done during each graph time step (3.3). While many variations to GAs, such as hybridizing with local search, seeding the initial population, or adapting the algorithm alongside the evolving solutions, have been shown to improve performance on graph coloring [2, 3], we seek to illuminate the relationship between the dynamic graph-coloring problem and GA performance, rather than determine the best possible algorithm. Consequently, we do not consider these improvements (or others), though we would recommend exploring these methods for real-world applications. Specifically, we focus on the relationship between the change rate of the graph, the amount of evolution done at each step, and the GAs' performance, as this relationship provides insight into the interaction between the two sources of dynamism: the dynamics of the problem and the dynamics of the algorithm.

2. METHODS

2.1 Graphs

We test our GAs on two different dynamic graph models. The first is a dynamic extension of $G(n,p)$ graphs developed by Erdős and Rényi [4]. In the static model, a $G(n,p)$ graph is a graph on n vertices where each edge appears with probability p. We define an extension of this model, $G(n,p,c_v)$ graphs, where c_v is the *vertex change rate*. To create a dynamic $G(n,p,c_v)$ graph, we start with a $G(n,p)$ graph for G_1, and at each successive time step, vertices in the graph G_t "die" (are removed in G_{t+1}) with probability c_v. To keep G at approximately n vertices, an expected nc_v vertices are added at each step. When a new vertex v is added, edges are added between v and each vertex already in the graph with probability p. Edges are only removed or added when their incident vertices are removed and added. Thus at any time t, G_t is a $G(n+\epsilon,p)$ graph, ϵ has high probability of being small, and an average of nc_v vertices change each step.

Our other dynamic graph model is a dynamic 2-D Euclidean graph. In a 2-D Euclidean graph, each vertex is assigned a random x and y coordinate in the unit square. Two

vertices are adjacent if the Euclidean distance between them is less than some α threshold. For the dynamic Euclidean graph model, we mirrored the $G(n, p, c_v)$ model, where n and c_v act as above. The parameter p is used to calculate α such that two random points in the unit square have a probability p being a distance of α apart and thus will result in a graph with an edge density p. As we only examine vertex-dynamic graphs, vertices do not move between time steps (as this would change edges) but are deleted and added at new random (x, y) locations at a rate controlled by c_v. For both graph models, we varied n, p, and c_v in order to investigate how these parameters affected the GA performance.

2.2 Algorithms

Throughout our tests, we compared the performance of three algorithms:

- DSATUR [1], the standard graph coloring heuristic algorithm that colors vertices in order of degree saturation (used as a baseline for performance)
- DGA, a genetic algorithm with a dynamic population (single population evolves as the graph changes)
- SGA, a genetic algorithm with a static population (new random population is generated each time step).

The only difference between the DGA and SGA was how the population was handled as the graph changed. For both, we used an steady-state genetic algorithm [18] with elitism where the two children produced by crossover and mutation replace the two individuals with the worst fitness, regardless of the fitness of the new children.

Fitness was calculated by using the standard greedy coloring on the permutation specified by the individual. Using just the number of colors in the decoding offers little gradation in the fitness landscape as many individuals decode with the same number of colors. To break ties between these individuals, we examined the number of vertices assigned each of the three least used colors; an individual with only a few vertices of a color seemed closer to using fewer colors than one with a even distribution of vertices among the colors. Thus, we calculated fitness as $f = n^3c + n^2c_1 + nc_2 + c_3$, where c is the number of colors used, c_1 is the number of vertices colored the least used color, c_2 is the number colored the second least used color, and c_3 is the number colored the third least used color. As we wanted to minimize the number of colors used, our GAs sought to minimize fitness.

Six standard crossover operators for permutation-based GAs are described in [17]. These are edge, order 1 (OX1), order 2 (OX2), position, partially mapped (PMX), and cycle crossover. As edge crossover preserves adjacency, which is not important for graph coloring, we did not implement it. We did implement the other five and in preliminary experiments, saw no significant difference in the performance of the GAs with different crossover operators. As such, we used OX1 in our main experiments as it performed reasonably well and is a standard crossover operator.

Similarly we considered three mutation operators (RAR, SWAP, inversion) described in [9]. We found that RAR and SWAP performed noticeably better than inversion and having no mutation, but observed no significant difference between RAR and SWAP. We chose to use the SWAP operator. Our other constant parameters were chosen with reasonable default values based on the authors' previous experience; fine-tuning is unlikely to alter the main qualitative results:

- Population size: 100
- Tournament selection with tournament size: 3
- Crossover rate: 0.7
- Mutation chance per individual: 0.5.

For the initial dynamic population and the static population at each step, we simply generated 100 random permutations on the vertices in the current graph. To repair each individual in the dynamic population each time the graph changed, we removed all vertices that had been removed from the graph, and then independently added the new vertices at random positions in each individual.

The graph parameters (n, p, c_v) and number of individuals created (and fitness evaluations done) e, are all parameters determined by the nature of the problem being solved and computational resources available. Thus, in order to study the difference between the DGA and SGA for a wide variety of problems, we performed a univariate sensitivity analysis, in turn varying each of n, p, c_v, and e from the following default parameter values:

- Graph size, n: 100
- Edge density, p: 0.6
- Change rate, c_v: .01
- Individuals evolved per step, e: 1000.

2.3 Experiment

In our experiments, each run consisted of generating a random graph and a random initial dynamic population, and then letting the graph change 150 steps. For each step:

1. Add a vertex with probability c_v. Repeat n times.
2. Add new vertices to the DGA's population.
3. Run DSATUR on the current graph.
4. Evolve the DGA's population for e individuals.
5. Create the SGA's population and evolve e individuals.
6. Record the number of colors used by DSATUR and the best individual from the DGA and SGA.
7. Remove each vertex with probability c_v.
8. Remove "dead" vertices from the DGA's population.

Each test consisted of working with a single dynamic graph for 150 steps—running all three algorithms on the same graph so that we could compare their performance. For each set of parameters, we performed 200 replicate runs and took the mean of the cumulative number of colors used by each of the algorithms for the entire run of the graph.

3. RESULTS AND DISCUSSION

As the only difference between the DGA and the SGA is how the population is maintained, the two genetic algorithms do the same amount of work when the parameters are kept constant. DSATUR, relatively speaking, requires very little computational effort. However, our experiments show that the genetic algorithms can outperform DSATUR for certain values of n, p, c_v, and e. Furthermore, as seen in [16], there are 3-chromatic graphs on $O(n)$ vertices where DSATUR will use n colors, while GAs can theoretically find an optimal solution. Finally, the SGA does not outperform the DGA, though there are some parameter ranges where the two have indistinguishable performance.

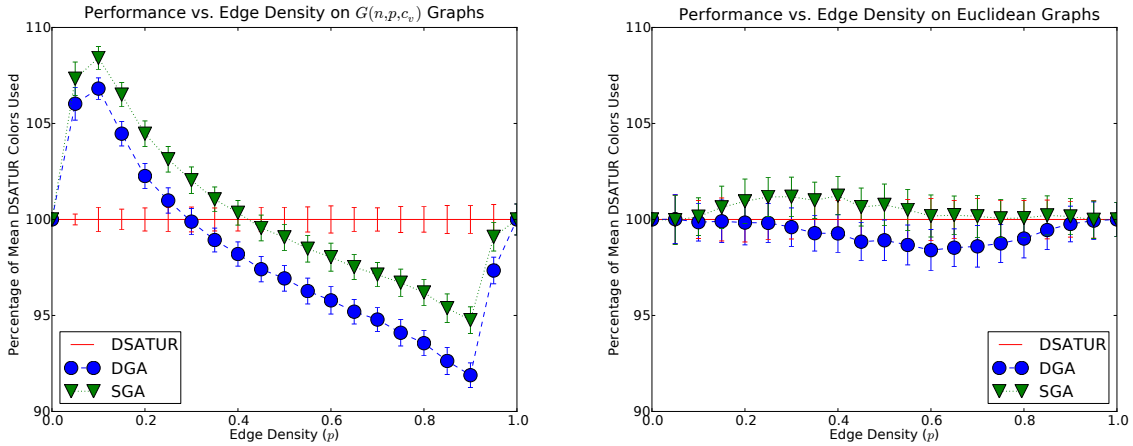

Figure 1: Performance of the genetic algorithms relative to DSATUR for varying edge densities, p, in $G(n, p, c_v)$ (left) and Euclidean (right) graphs. We can see that the DGA always outperformed SGA, and sometimes outperformed DSATUR, depending on edge density and graph structure.

In the figures that follow, we give performance values as a percentage of the mean number of colors DSATUR used so we can easily compare relative performance of the algorithms. Better performance occurs when the GAs use *fewer* colors relative to DSATUR. The error bars represent 95% confidence intervals estimated from the 200 replicate runs at each parameter setting and are scaled by the DSATUR mean. Despite the same number of replicate runs, the confidence intervals are much wider for the Euclidean graphs than for the $G(n, p, c_v)$ graphs, suggesting Euclidean graphs have a wider range of chromatic number than $G(n, p, c_v)$ graphs with the same parameters.

3.1 Edge Density (p)

As shown in Figure 1, the relative performance of the GAs to DSATUR depends strongly on p for $G(n, p, c_v)$ graphs. For $G(n, p, c_v)$ graphs with edge densities between .1 and .95, the relative genetic algorithm performance increases as edge density increases. Additionally, on $G(n, p, c_v)$ graphs, the DGA performance relative to SGA does not seem to depend on p, as there is a fairly consistent gap between their performance levels for almost all values of p.

For Euclidean graphs, the relative performance of the genetic algorithms to DSATUR is much less dependent on p. The DGA tends to slightly outperform DSATUR, while the SGA lags behind DSATUR. Higher edge densities in Euclidean graphs are likely to form cliques or near-cliques, which are colored effectively by prioritizing by degree saturation, and so DSATUR is able to do just as well as the GAs. However, in $G(n, p, c_v)$ graphs, the additional edges are more spread out through the entire graph, making graphs with higher edge densities harder to color. This provides the genetic algorithms more potential to improve on the DSATUR coloring, leading to a smaller percentage of colors used.

3.2 Change Rate (c_v) & Evolution Per Step (e)

The change rate of the graph, c_v, and the amount of evolution done per step, e, present an interesting relationship between the dynamic nature of the problem and the dynamic nature of the algorithm. As shown in Figure 2, more evolution leads to better performance of the GAs relative to DSATUR, especially on $G(n, p, c_v)$ graphs. This is not surprising since more evolution means more work is being done. However, on Euclidean graphs, more evolution (beyond $e \approx 1500$) does not substantially help the GAs improve its performance relative to DSATUR. Again, this suggests that DSATUR is able to get close to the optimal coloring on Euclidean graphs, whereas on the $G(n, p, c_v)$ DSATUR's coloring may be further from optimal, and more evolution allows the GAs to improve more compared against it.

In Figure 3, we observe that under constant evolution, the SGA and DSATUR performance differ by a constant percentage regardless of change rate. This is expected as the SGA does not interact with the changing graph and therefore its performance should not depend on the change rate. However, the DGA presents a very different picture. At low change rates, the DGA is able to noticeably outperform the SGA because small changes to the graph do not force substantial changes to the population and thus evolution can continue with minor disruption across graph changes. At higher change rates, there is much more upheaval in the dynamic population and so it offers less of an advantage over the SGA. Because new vertices are added to the individuals of the dynamic population randomly, the dynamic population is much closer to the fresh population of the SGA when there are lots of new vertices. Thus, the DGA starts to act more like the SGA as we have a more dynamic problem.

At very low levels of evolution, the DGA clearly offers better performance because the effects of evolution are allowed to accumulate. The SGA, on the other hand, starts with a new random population at every step, and thus does not have enough evolution to find a good solution. In contrast, at higher levels of evolution, the DGA performance only outperforms the SGA at very low change rates because with more evolution, the SGA has time to catch up with the evolution accumulated by the DGA.

While these trends are present in both $G(n, p, c_v)$ and Euclidean graphs, the gap between the DGA and SGA is much

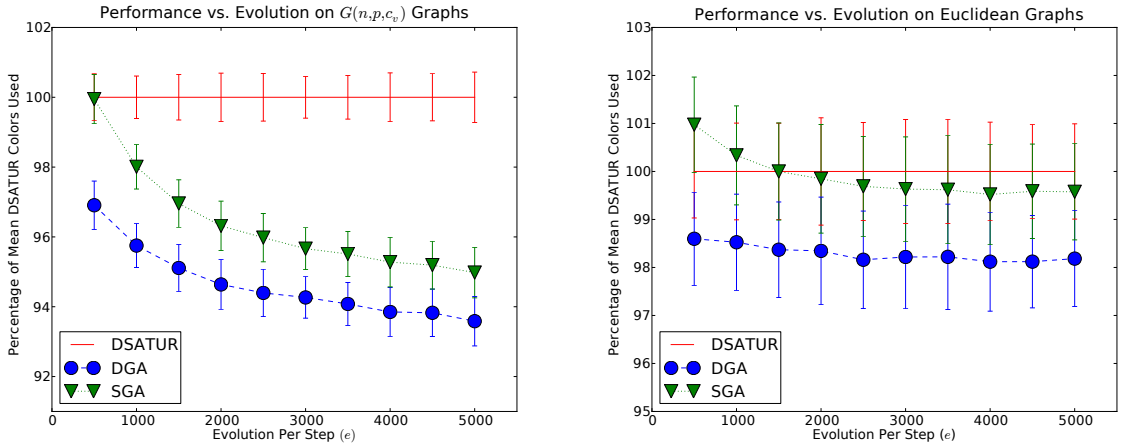

Figure 2: Performance of the GAs relative to DSATUR for varying evolution per step, e, in $G(n, p, c_v)$ (left) and Euclidean (right) graphs. As usual with GAs, there are diminishing returns on the GA performance as the amount of evolution done is increased. On $G(n, p, c_v)$ graphs, increasing evolution leads to better relative performance of the GAs. On Euclidean graphs, however, the benefit of more evolution levels off much faster.

wider for Euclidean graphs, and the DGA is better able to outperform the SGA. In a Euclidean graph, the edges are clustered locally and it is rare for a new vertex to connect two different sections of the graph. Thus changes in the graph are felt locally and we hypothesize that a good coloring is therefore more easily carried forward from the time step before. On $G(n, p, c_v)$ graphs, however, a new vertex can cause major disruptions in the coloring from the step before, and thus there is likely less benefit in keeping a dynamic population. This suggests that whenever changes are local rather than widespread, a dynamic population is better able to provide performance increases.

3.3 Individuals per Vertex

From our experiments with e and c_v, it is clear that problems the allow more evolution per graph time step and have low change rates are more solvable with dynamic GAs than problems with higher change rates or problems where the amount of evolution per graph time step is necessarily limited. For practical applications, however, it is likely that the amount of evolution is limited by the availability of computational resources and that the change rate is dictated by the how fast the situation being modeled is changing.

For example, we could imagine that we have 4 new tasks and the computational resources to evolve 1000 individuals per minute. We might ask: is it better to change 4 vertices every minute and evolve for 1000 fitness evaluations between changes, or is it better to change 1 vertex every 15 seconds and evolve for 250 fitness evaluations between each change? In the former case we have more evolution per step but a higher change rate, and in the latter case we have a lower change rate but less evolution between changes. Which will offer better performance?

To examine this question of trade-offs, we define a new parameter, $e_v = \frac{e}{nc_v}$, which represents the number of fitness evaluations *per changed vertex*. We then ran the same experiments as before on $G(n, p, c_v)$ graphs for various e_v values at different levels of evolution. The values for nc_v needed to achieve a given e_v with a given e value are given in Table 1. Instead of running for 150 time steps, we ran these graphs until 200 new vertices were added so all runs would process the same number of vertices. We examine the performance of the DGA at evolution rates of 500, 1000, 1500, and 2000 against DSATUR for varying e_v levels.

		e			
		500	1000	1500	2000
e_v	100	5	10	15	20
	200	2.5	5	7.5	10
	300	$\frac{5}{3}$	$\frac{10}{3}$	5	$\frac{20}{3}$
	400	1.25	2.5	3.75	5
	500	1	2	3	4

Table 1: nc_v values (the expected number of changed vertices each step) determined by e and e_v.

As seen in Figure 4, for this problem domain it is clearly better to choose longer time steps with more evolution and consequently higher change rates when possible, even though the DGA performance is generally better at lower change rates. This suggests that the effect of evolution is stronger than the effect of change rate in determining the performance of the GA. Frequent small changes are harder for the genetic algorithm to deal with than infrequent drastic ones, indicating that the time after a change that the genetic algorithm needs to adjust to the new problem is not linearly proportional to how much the graph has changed. It is better to change the problem as infrequently as possible so that you can maximize the amount of uninterrupted evolution.

3.4 Graph Size (n)

Finally, we experimented with the size of the graph n, while maintaining a constant amount of evolution ($e = 1000$). As seen in Figure 5, the GAs only outperform DSATUR for graphs with fewer than 200 vertices, and their performance

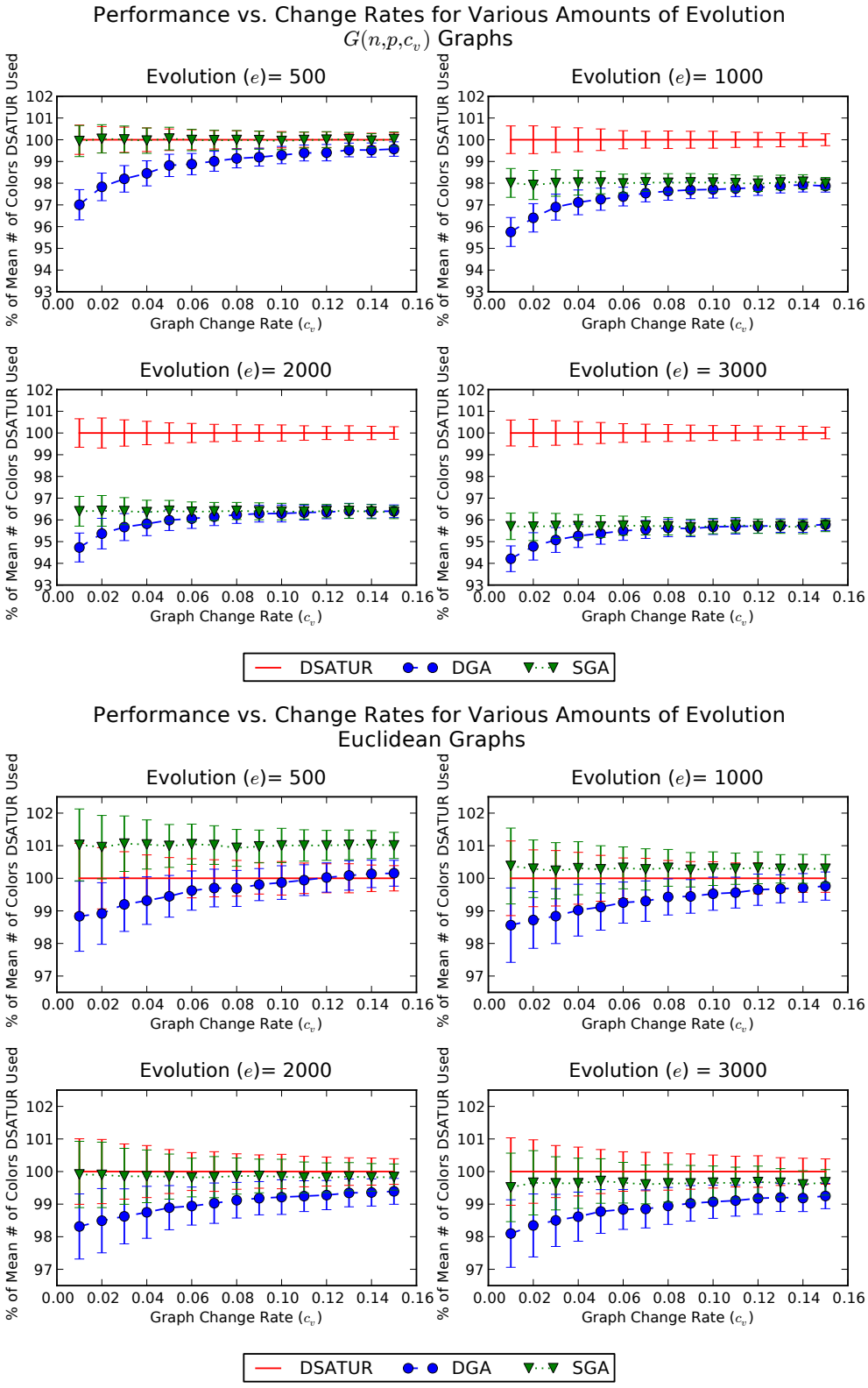

Figure 3: Performance of the genetic algorithms relative to DSATUR for varying change rates, c_v, and evolution, e, in $G(n, p, c_v)$ (top) and Euclidean (bottom) graphs. As change rates increase, DGA performance converges to SGA performance. This happens faster at higher rates of evolution. On Euclidean graphs, there is a much wider gap between DGA and SGA performance.

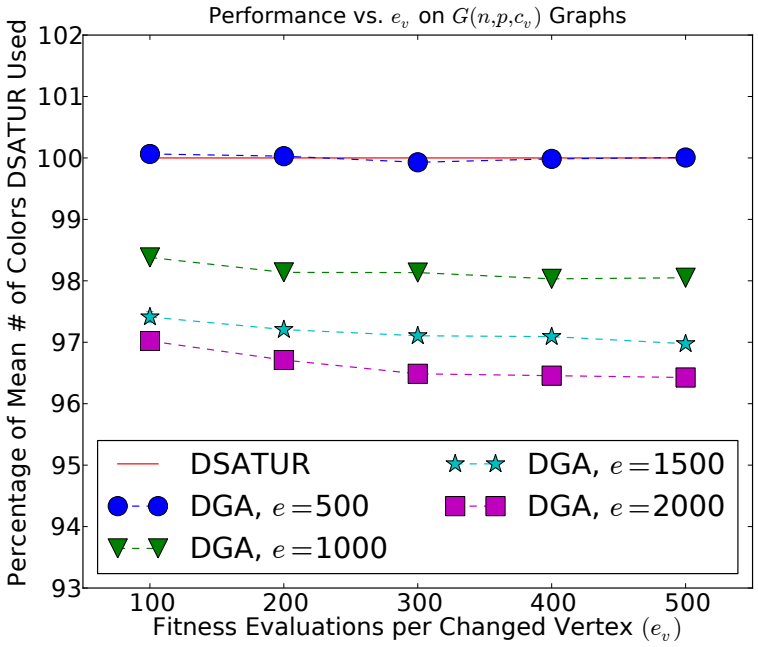

Figure 4: Performance of DGA relative to DSATUR at different amounts of evolution per step, e, for varying rates of individuals per changing vertex. Specific values for the expected number of changed vertices (nc_v) for a given e, e_v point are given in Table 1, but in general, higher levels of evolution at a fixed number of individuals per changed vertex require a higher number of changed vertices. Regardless of e_v, higher rates of evolution (and consequent higher change rates) have stronger relative performance to DSATUR.

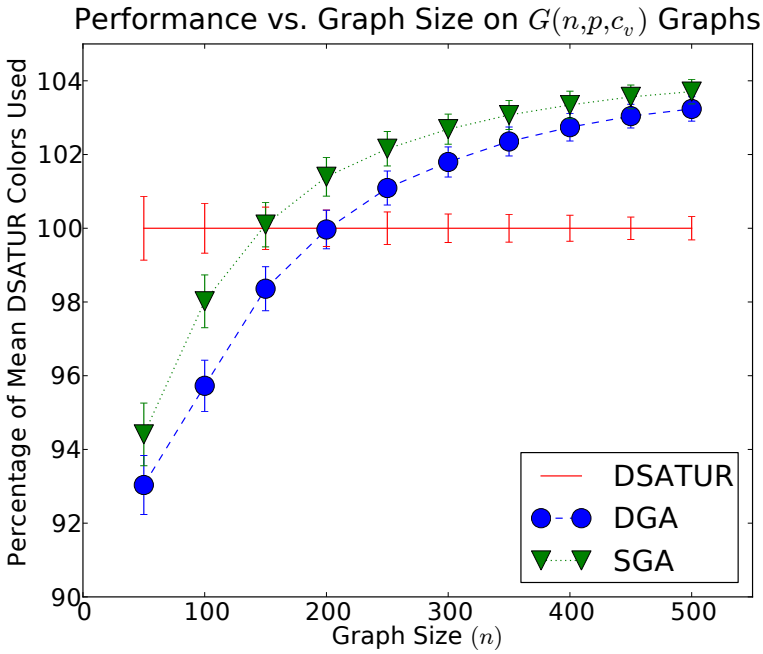

Figure 5: Performance of the genetic algorithms relative to DSATUR for varying sizes of $G(n, p, c_v)$ graphs. As graph sizes increase, the genetic algorithms' performance relative to DSATUR decreases. However, this trend levels off as graph sizes continue to increase.

relative to DSATUR decreases as the graph size increases. This suggests that a GA approach may not scale well to very large dynamic graphs. However, the worsening performance appears to be leveling off as graph sizes continue to increase, which is a good sign. It may also be possible to outperform DSATUR even at larger graph sizes if the GA is given sufficient computational resources. Furthermore, the dynamic graph coloring problem is still challenging for graphs with fewer than 200 vertices, so even if GA techniques do not scale well to large graphs, they could still serve many practical applications that fall into this size range.

4. CONCLUSIONS AND FUTURE WORK

With this work, we first described different variations of the dynamic graph coloring problem and defined two different dynamic graph models that extend classical random graph models. We then explored the effect of several different parameters on the performance of a genetic algorithm with a dynamic population and a genetic algorithm that starts with a new population at every step relative to DSATUR for the dynamic graph coloring problem. We are able to demonstrate conditions where a dynamic population is able to offer significant performance increases relative to both a genetic algorithm with a static population and to the standard graph coloring algorithm, DSATUR. Additionally, the added complexity/overhead of keeping a dynamic population while the graph changes is insignificant with a permutation based GA. In all circumstances, the dynamic population GA performed at least as well as the static population GA, and sometimes offered much better performance.

Broadly, this work supports the idea that the more dynamic a problem is, an algorithm is less able to utilize information from earlier time steps of the problem as these steps are less similar to the current problem. Then, highly dynamic problems reduce to a succession of static problems, while slightly dynamic problems are better tackled by a more dynamic algorithm. Future work is needed to explore graphs that are edge-dynamic, or both edge- and vertex-dynamic, as well as graphs with different structures from the two classes ($G(n, p, c_v)$ and Euclidean) examined here.

As mentioned early in the paper, some of the most successful static graph coloring algorithms use hybrid approaches, combining GAs with other heuristics or local search mechanisms, or use partition-based genetic algorithms. Since this paper presents the first application of genetic algorithms for the dynamic graph coloring problem, we chose to keep our genetic algorithms relatively pure and straightforward, but it would be interesting for follow-up papers to explore how much benefit could be gained through hybrid techniques. Further work also needs to be done in adapting more advanced static genetic algorithms to the problem of coloring a dynamic graph, There are also many variations of the graph coloring program that correspond to different practical applications, and future work could investigate if the techniques outlined here are applicable to dynamic versions of these problems.

5. REFERENCES

[1] Brélaz, D. New methods to color the vertices of a graph. *Communications of the ACM 22*, 4 (Apr. 1979), 251–256.

[2] Davis, L. *Handbook of genetic algorithms*. VNR computer library. Van Nostrand Reinhold, 1991.

[3] Eiben, A., van der Hauw, J., and van Hemert, J. Graph coloring with adaptive evolutionary algorithms. *Journal of Heuristics 4* (1998), 25–46. 10.1023/A:1009638304510.

[4] Erdős, P., and Rényi, A. On the evolution of random graphs. In *Publication of the Mathematical Institute of the Hungarian Academy of Sciences* (1960), pp. 17–61.

[5] Galinier, P., Hamiez, J.-P., Hao, J.-K., and Porumbel, D. Recent advances in graph vertex coloring. In *Handbook of Optimization*, I. Zelinka, V. Snášel, and A. Abraham, Eds., vol. 38 of *Intelligent Systems Reference Library*. Springer Berlin Heidelberg, 2013, pp. 505–528.

[6] Galinier, P., and Hao, J.-K. Hybrid evolutionary algorithms for graph coloring. *Journal of Combinatorial Optimization 3*, 4 (1999), 379–397.

[7] Garey, M. R., and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[8] Harary, F., and Gupta, G. Dynamic graph models. *Mathematical and Computer Modelling 25*, 7 (1997), 79–88.

[9] Hauw, K. v. d. Evaluating and Improving Steady State Evolutionary Algorithms on Constraint Satisfaction Problems. Master's thesis, Leiden University, 1996.

[10] Holland, J. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press, 1975.

[11] Jin, Y., and Branke, J. Evolutionary optimization in uncertain environments-a survey. *Trans. Evol. Comp 9*, 3 (June 2005), 303–317.

[12] Lovász, L., Saks, M. E., and Trotter, W. T. An on-line graph coloring algorithm with sublinear performance ratio. *Discrete Mathematics 75*, 1-3 (1989), 319–325.

[13] Marx, D. Graph Coloring Problems and Their Applications in Scheduling. In *in Proc. John von Neumann PhD Students Conference* (2004), pp. 1–2.

[14] Preuveneers, D., and Berbers, Y. Acodygra: an agent algorithm for coloring dynamic graphs. In *Symbolic and Numeric Algorithms for Scientific Computing* (September 2004), vol. 6, pp. 381–390.

[15] Shirinivas, S., Vetrivel, S., and Elango, N. Applications of Graph Theory in Computer Science An Overview. *International Journal of Engineering Science and Technology 2*, 9 (2010), 4610–4621.

[16] Spinrad, J. P., and Vijayan, G. Worst case analysis of a graph coloring algorithm. *Discrete Applied Mathematics 12*, 1 (1985), 89–92.

[17] Starkweather, T., and of Computer Science, C. S. U. D. *A Comparison of genetic sequencing operators*. No. 106 in Technical report (Colorado State University. Dept. of Computer Science). Colorado State University, Department of Computer Science, 1991.

[18] Syswerda, G. A study of reproduction in generational and steady-state genetic algorithms. *Foundation of Genetic Algorithms* (1991), 94–101.