Comparing Evolutionary Algorithms for the Graph Colouring Problem

Daniel Marshall
Trisha Goel
Robert Dennison
Sam Clowes

28th May 2020

Contents

Chapter 1

Introduction

The graph colouring problem involves labelling the vertices of a graph with the minimum possible number of colours such that no adjacent vertices share the same colour. It turns out that many practical real world problems can be reduced to graph colouring, including class scheduling, register allocation in compilers, frequency assignment in mobile networks, air traffic management, and many others.

In spite of its many applications, the graph colouring problem is difficult to solve efficiently. It was proven in the 1970s [1] that the graph colouring problem is NP-complete, which means that assuming P $\neq$ NP it is not possible to find an algorithm which optimally colours graphs in polynomial time. However, one possible technique which could be considered for solving this problem is the application of evolutionary algorithms. These algorithms use mechanisms inspired by biological evolution, where candidate colourings play the role of individuals in a population, a fitness function is used to determine the quality of the colourings, and the population evolves through processes such as crossover, mutation and replacement.

Given a graph $G(V, E)$, the graph colouring problem requires finding the minimal number of colours χ (the chromatic number) such that there exists a vertex colouring (using χ colours) with no adjacent vertices of the same colour (with no conflicts). The chromatic number can be determined algorithmically by starting from a value k that is certainly larger than χ (for example, $k = |V|$ will always be enough), finding a k-colouring without conflicts, then decrementing k and repeating the process. This problem will become increasingly difficult until it can no longer be solved by the given algorithm.

Though evolutionary algorithms have often been successfully used in practice for graph colouring, there is still little understanding of when and why they are effective. Some theoretical results have been shown for simple algorithms on well-structured graphs, but the importance of crossover and more complex mutation operators on an algorithm's behaviour is still poorly understood even in these well-structured cases, and similarly there is little knowledge of how even basic algorithms behave when confronted with more challenging classes of graphs such as those which result from practical problems in the real world.

Our project aims to design and implement a piece of software to compare the performance of evolutionary algorithms for the graph colouring problem. The eventual goal is to bridge the gap between the existing theoretical understanding of simple algorithms and the more practical but more complex algorithms that can be found in the literature. We will test both state-of-the-art and simple theory-driven algorithms on both practical benchmark graph instances and basic graphs with mathematical definitions that are easier to state, which we hope will give a clearer picture of how these algorithms perform in a variety of cases. We hope to also gain some understanding of how the differing behaviour of these algorithms affects how they perform, and thus which aspects of evolutionary algorithm design are most important to consider when attempting to solve the graph colouring problem as efficiently as possible.

Chapter 2

Literature Review

There are many different types of evolutionary algorithms. For this project, simple evolutionary algorithms such as $(1+1)$ EA and Randomised Local Search (RLS) will be compared against complex and state-of-the-art algorithms for the graph colouring problem. As previously stated, evolutionary algorithms are made up of components that are inspired by biological evolution. This usually includes crossover, mutation and replacement functions. Complex algorithms often also make use of local search algorithms such as Tabu Search [2] to improve the offspring k-colouring by minimising the number of conflicts. The number of conflicts in a graph colouring problem is used to describe the number of vertices that share an edge with a vertex of the same colour.

To solve the k-colouring problem for a graph $G = (V, E)$ we consider all possible partitions of V into k colour classes, including partitions which are not proper k-colourings[1]. Each partition is scored equal to the number of conflicts in the k-colouring, where a score of 0 corresponds to a partition which is a proper k-colouring. The k-colouring problem can be solved by minimizing this scoring function.

2.1 (1+1) EA and RLS

The $(1+1)$ EA and RLS algorithms rely on mutation and replacement operators to find the optimal k-colouring for a graph. In $(1+1)$ EA shown in Algorithm 1, a vertex of the graph colouring is mutated with a probability of $\frac{1}{n}$ by changing its colour randomly to any colour other than the one it currently is. This is carried out on all the vertices to form a new colouring. The colouring replaces the previous one if it has fewer conflicts.

Algorithm 1: $(1+1)$ EA

while *optimum not found* **do**
  Generate y by mutating each component x_i with probability $1/n$, if mutating, choose a new value $y_i \in \{1, \ldots, k\} \setminus \{x_i\}$
  If y has no more conflicts than x, let $x = y$
end

The RLS algorithm shown in 2 follows a similar approach where the new colouring is obtained by randomly choosing a new colour for one vertex of the graph while keeping the remaining colours the same. If the new colouring has fewer conflicts than the previous, it replaces the previous colouring. Both algorithms continue until an optimal k-colouring is found for the graph.

Algorithm 2: Randomised Local Search (RLS) Algorithm

while *optimum not found* **do**
  Generate y by choosing an index $i \in \{1, \ldots, n\}$ uniformly at random, choosing a new value $y_i \in \{1, \ldots, k\} \setminus \{x_i\}$ uniformly at random and setting $y_j = x_j$ for all $j \neq i$
  If y has no more conflicts than x, let $x = y$
end

[1] A proper k-colouring is one where V is partitioned so that no adjacent vertices belong to the same colour class.

2.2 (2+2) GA with fitness sharing

The $(2 + 2)$ genetic algorithm with fitness sharing [3] is another algorithm we will evaluate. The algorithm (Algorithm 3) uses a population size of two individuals and a two-point crossover function to produce two offspring.

Crossover: The crossover function selects two crossover points from the two parents and then swaps the colour classes of the vertices between the two points. Binary encoding is usually used to represent the vertices, so each vertex would be assigned a value of 0 or 1. Each point is flipped from one value to the other with a probability of $1/n$, and each bit is flipped independently of the others.

Fitness Sharing: Fitness sharing reduces an individual's fitness score by considering how similar it is to other members of the population. The equation used to calculate the similarity between two individuals x and y is $S(x, y) = \max\{0, 1 - d(x, y)/\sigma\}$, where σ represents the sharing distance. The sharing distance is the maximum distance that two individuals require between them to share a fitness. The equation used to calculate the shared fitness of an individual x is $f(x, P) = f(x)/S(x, P)$ where $f(x)$ is the fitness of individual x and $S(x, P) = \sum_{y \in P} S(x, y)$. The shared fitness across the whole population can be calculated using $f(P) = \sum_{x \in P} f(x, P)$ which is the sum of the shared fitness values between each individual and the population P.

After the crossover function is applied, each vertex of the offspring is mutated with a probability of p to swap its colour class. The two best individuals are selected from the parents and offspring to replace the population. This is achieved by selecting two individuals from the population P' that combine to give the greatest shared fitness value $f(P)$.

Algorithm 3: (2+2) GA with fitness sharing

Initial population P consists of two individuals chosen independently and uniformly at random
Selection: Both individuals x and y are chosen
if $p > \frac{1}{2}$ **then**
  $x' = mutate(x)$
  $y' = mutate(y)$
else
  $(\tilde{x}, \tilde{y}) = crossover(x, y)$
  $x' = mutate(\tilde{x})$
  $y' = mutate(\tilde{y})$
end
$P' = P \cup \{x', y'\}$
Replacement: Choose population $P \subseteq P'$ of size 2 with the maximal $f(P)$ value

2.3 Iterated Local Search

The Iterated Local Search algorithm (Algorithm 4) in [4] uses a Kempe chain or colour elimination mutation operator and Grundy Local Search to produce a new colouring for a graph.

Algorithm 4: Iterated Local Search Algorithm

Create initial colouring by choosing for each vertex v a colour in $\{1, \ldots, deg(v) + 1\}$ independently and uniformly at random
Replace x by the result of Grundy local search applied to x
repeat forever
  Let y be the result of a mutation operator applied to x
  Let z be the outcome of Grundy Local Search applied to y
  if $z \succeq x$ **then**
    $x = z$
  end

Kempe Chain Operator: The graph is mutated by applying a Kempe chain operator as shown in Algorithm 5. The graph may contain paths consisting of vertices coloured using two alternating colours. The Kempe chain operator is applied to vertices in such paths to swap the colour of each vertex. This is done by first selecting a vertex and randomly changing it to a colour j. The colours i or j of all vertices in the connected component $H_j(v)$ are then swapped to give the opposite colour. $H_j(v)$ is a connect component of the graph where all vertices are coloured $c(v)$ or j and contains the vertex v. This operation does not produce

any conflicting vertex colours as each neighbouring vertex is swapped to the opposite colour. Once the graph has been mutated, the Grundy Local Search algorithm is applied.

Algorithm 5: Kempe Chain Operator

```
Choose v ∈ V and j ∈ {1, ..., deg(v) + 1} uniformly at random
for all u ∈ H_j(v) do
    if c(u) = c(v) then
        c(u) = j
    else
        c(u) = c(v)
    end
end
```

Colour Elimination: Another mutation operator is the colour elimination algorithm (Algorithm 6). It aims to reduce the colour value of a vertex by selecting a smaller value from its neighbourhood, and replacing all instances of them with a different colour. The eliminated colour is then used to colour the original vertex.

Algorithm 6: Colour Elimination Algorithm

```
Choose v ∈ V uniformly at random
if c(v) ≥ 3 then
    Choose i, j ∈ {1, ..., c(v) − 1}, i ≠ j, uniformly at random
    Let v_1, ..., v_l enumerate all i-coloured neighbours of v
end
for all u ∈ H_j(v_1) ∪ ... ∪ H_j(v_l) do
    if c(u) = i then
        c(u) := j
    else
        c(u) := i
    end
end
```

Grundy Local Search: The Grundy Local Search [5] algorithm (Algorithm 7) is used to give a new Grundy colouring for the graph. A Grundy colouring's vertices are all assigned the smallest possible colour class and a Grundy vertex v is assigned the smallest colour that is not assigned to any of its neighbours $N(v)$. The algorithm selects colours to assign to each vertex, such that every vertex is made a Grundy vertex.

Algorithm 7: Grundy Local Search

```
while the current colouring is not a Grundy colouring do
    Choose a non-Grundy vertex v
    Set c(v) = min{i ∈ {1, ..., n} | ∀w ∈ N(v) : c(w) ≠ i}
end
```

The replacement criteria for the colouring is that the new colouring should contain a greater number of vertices in smaller colour classes. The replacement criteria for the colouring is based on two conditions. Firstly, the number of conflicts is compared between the new colouring z and the previous colouring x. If z has fewer conflicts it is selected for replacement. If both z and x have the same number of conflicts, the number of vertices of each colour value is compared. This is done by comparing the number of vertices with the highest colour value. If z has fewer than x it replaces it. If the numbers are the same, the colour value is decreased until a difference is found.

2.4 The Hybrid Colouring Algorithm

A hybrid evolutionary algorithm consists of a combination of a local search function and a highly specialized crossover function. The crossover function is used to produce a new set of offspring which are then improved using the local search function. The improved offspring are then what is inserted back into the population. This particular algorithm was presented by Galinier & Hao in [6].

InitPopulation: HCA uses a greedy saturation algorithm modified to produce a partition of V into k classes. Firstly, start with k empty colour classes $V_1 = \ldots = V_k = \phi$. A vertex $v \in V$ is chosen such that v has

Algorithm 8: Hybrid Colouring Algorithm

Input : graph $G = (V, E)$, integer k
Output: the best configuration ever found
$Pop =$ **InitPopulation**(|Pop|)
while *a* stopping condition *is not met* **do**
  $(I_1, I_2) =$ **SelectParents**(Pop)
  $i =$ **Crossover**(I_1, I_2)
  $i =$ **LocalSearch**$(i,$ L$)$
  $Pop =$ **UpdatePopulation**(Pop, i)
end

the minimal number of allowed classes. We then assign v to a colour class, choosing from the allowed classes of v a class V_i such that i is minimized. This algorithm cannot assign classes to all vertices, so the remaining unassigned vertices are randomly assigned a colour class V_i.

SelectParents: In the HCA, two parents I_1, I_2 are chosen at random from the population Pop.

Crossover: HCA uses the GPX crossover function. This function works to build the colour classes $V_1, \ldots, V_k$ for the offspring one at a time. For class V_l it chooses a parent depending on whether l is odd or even. It then chooses the parent class containing the largest number of vertices to become class V_l. The set of vertices contained in V_l are then deleted from both parents. This is repeated for all classes and then any unassigned vertices are randomly allocated a class.

Algorithm 9: HCA's GPX Crossover Function

Input : parents $I_1 = V_1^1, \ldots, V_k^1, I_2 = V_1^2, \ldots, V_k^2$
Output: offspring $i = V_1, \ldots, V_k$
for $l = 1$ **to** k **do**
  if l is odd let $A = 1$, else $A = 2$
  choose i such that V_i^A has maximum cardinality
  $V_l = V_i^A$
  remove vertices of V_l from I_1, I_2
end
Randomly assign remaining vertices $V - (V_1 \cup \ldots \cup V_k)$

LocalSearch: This is used to improve the offspring generated by the crossover algorithm before inserting the offspring into the population. Whilst any local search algorithm can be used, the HCA described uses a Tabu Search method.

Algorithm 10: HCA's Tabu Search Function

Input : graph $G = (V, E)$, configuration i_0
Output: best configuration found i
$i = i_0$
while *a* stopping condition *is not met* **do**
  choose best authorised move $< v, s >$
  introduce couple $< v, i(v) >$ into Tabu list for t_l iterations
  perform move $< v, s >$ in i
end

UpdatePopulation: To insert the new offspring i into the population, the worst of the two parents I_1, I_2 is replaced by i.

Galinier & Hao found that the HCA outperforms the Tabu Search Algorithm when run on graphs from the DIMACS graph set. When finding k-colourings, the HCA finds solutions quicker than TS. As well as this, when k is left undefined, HCA finds better solutions for graphs than TS (colourings with fewer colours, smaller k).

2.5 Evocol

Another of the algorithms we would like to analyse is Evocol, which was presented in [7] and is a hybrid evolutionary algorithm based on two simple (but effective) ideas. First, the algorithm uses an enhanced crossover

operator that collects the best colour classes out of more than two parents, where the classes are ranked based on fitness and size. Second, the algorithm uses distances to ensure population diversity, by maintaining distances between individuals in the population which are as large as possible whenever the population changes.

A skeleton of the Evocol algorithm is given in Algorithm 11, where the stopping condition is either to find a legal colouring or to reach a predefined time limit.

Algorithm 11: Evocol

input : the search space Ω
result: the best configuration ever found
randomly initialise parent population $Pop = (I_1, I_2, \ldots, I_{|Pop|})$
while *a* stopping condition *is not met* **do**
  for $i = 1$ **to** p *(p =number of offspring)* **do**
    repeat
      $(I_1, I_2, \ldots, I_n) =$ **SelectParents**(Pop, n)
      $O_i =$ **Crossover**$(I_1, I_2, \ldots, I_n)$
      $O_i =$ **LocalSearch**(O_i, maxIter)
    until ***AcceptOffspring****(Pop, O_i)*
  end
  $Pop =$ **UpdatePopulation**$(Pop, O_1, O_2, \ldots, O_p)$
end

SelectParents: Evocol's parent selection simply consists of choosing n different individuals uniformly at random from the population Pop.

Crossover: This algorithm uses a multi-parent crossover operator (MPX) which collects in the offspring the best colour classes from several parents. Each class in each parent receives a score based on the number of conflicts and the size of the class. Crossover proceeds as shown in Algorithm 12.

Algorithm 12: Evocol's multi-parent crossover MPX

input : parents $I_1, I_2, \ldots, I_n$
result: offspring O
O = empty, i.e. start with no vertex colour assigned
for *currentColour* $= 1$ **to** k **do**
  foreach *parent* $I_i \in \{I_1, I_2, \ldots, I_n\}$ **do**
    foreach *colour class* I_i^j *in* I_i **do**
      remove from I_i^j all vertices already assigned in O
      conflicts $= |\{(v_1, v_2) \in I_i^j \times I_i^j : (v_1, v_2) \in E\}|$
      classSize $= |I_i^j|$
      score$[I_i^j]$ = conflicts $\times$ $|V|$ - classSize
    end
  end
  set $(i^*, j^*) = \text{argmin}_{(i,j)}$ score $|I_i^j|$
  foreach $v \in I_{i^*}^{j^*}$ **do**
    $O[v]$ = currentColour
  end
end
foreach *unassigned* $v \in O$ **do**
  $O[v]$ = a colour that generates the least number of conflicts
end

LocalSearch: The local search procedure used by Evocol is a modified version of the classical Tabu Search algorithm often used for graph colouring, known as Tabucol. [2] The algorithm iteratively moves from one colouring to another by modifying the colour of a conflicting vertex until either a valid colouring is found or a predefined maximum number of iterations is reached. Each performed move is marked 'Tabu' for a number of iterations T_l, so a move cannot be re-performed that has been performed in the last T_l iterations. In the case of Evocol, $T_l = \alpha * f(C) + random(A) + \lfloor \frac{M}{M_{max}} \rfloor$, where $\alpha = 0.6$, $A = 10$ (and $random(A)$ is a random integer in $[1 \ldots A]$), M is the number of the last consecutive moves that kept the fitness function constant, and M_{max} = 1000.

The diversity control used by Evocol is based on a distance metric defined using the partition colouring representation of graphs. This representation says that a colouring is a partition of V into distinct classes of vertices with given colours. The distance between two partitions under this metric is the minimum number of elements that need to be moved between classes of the first partition so that it becomes equal to the second partition.

AcceptOffspring: Evocol inserts the offspring in the population only if its distance to each existing individual is greater than a predefined threshold R. Thus, the offspring acceptance procedure will either reject an offspring O that is less distance than R away from an individual I, or replace I with O if $f(O) \leq f(I)$ (i.e. if O is better than I).

UpdatePopulation: This procedure determines which existing individual is eliminated for each offspring that needs to be inserted, taking into account fitness values and population diversity. The procedure encourages elimination of individuals that are too close to other individuals in order to encourage diversity. It works as shown in Algorithm 13.

Algorithm 13: Evocol's replacement (elimination) function

```
input  : population Pop = (I_1, I_2, ..., I_|Pop|)
output: the individual to be eliminated
repeat
 |  C_1 = RandomIndividual(Pop)
until AcceptCandidate(C_1)
minDist = maximum possible integer
foreach I ∈ Pop − {C_1} do
 |  if d(I, C_1) ≤ minDist then
 |  |  if AcceptCandidate(I) then
 |  |  |  minDist = d(I, C_1)
 |  |  |  C_2 = I
 |  |  end
 |  end
end
if f(C_1) ≤ f(C_2) then
 |  return C_2
else
 |  return C_1
end
```

The **AcceptCandidate** function always accepts a candidate C_i for elimination if it has a fitness value lower than the median, and accepts C_i only with 50% probability if C_i has a fitness value higher than the median. Only the best individual is fully protected; it can never become a candidate for elimination unless more than half of the population is made up of best individuals.

Evocol was presented in the literature with the following parameter values: $|Pop| = 15$ (population size), $n = 3$ (number of parents), $p = 3$ (number of offspring constructed each generation), $maxIter = 100000$ (the maximum number of iterations of the Tabu Search procedure), $R = 10\%|V|$ (the sphere radius, the minimum imposed distance between two individuals in the population).

2.6 Evo-Div

The Evo-Div algorithm proposed in [8] is shown in Algorithm 14 and uses a combination of crossover, local search, mutation and replacement operators to produce a new graph colouring. A k-colouring is composed of a set of disjoint subsets of the vertices with each subset corresponding to a different colour class. The population consists of a set of colouring configurations for the graph which is used to select parent configurations for the crossover function. It also has a multi-parent crossover function that combines two or more colourings from the population to form an offspring by selecting the best colour classes. A local search algorithm is then run on the offspring to improve the colouring and remove conflicts. The fitness function is used to count the number of conflicts in a colouring. This uses a modified version of the Tabucol [2] algorithm and runs for a set number of maximum iterations. Mutation is used to maintain the population diversity and applied if the offspring is rejected for replacement more than a set number of times. This prevents offspring too similar to other population members from being added to the population.

WIPX-Crossover: The algorithm (Algorithm 15) uses a multi-parent crossover function called the Well Informed Partition Crossover to select the best colour classes from the parents to add to the offspring. Each

Algorithm 14: Evo-Div algorithm

input : a graph $G = (V, E)$ and a positive integer k
output: the best fitness value ever reached
Initialise (randomly) parent population $Pop = \{I_1, I_2, \ldots, I_{|Pop|}\}$
while *stopping condition is not met* **do**
  rejections = 0
  repeat
    $(I_1, I_2, \ldots, I_n) =$ **RandomParents**(Pop, n)
    $O =$ **WIPX-Crossover**$(I_1, I_2, \ldots, I_n)$
    if *rejections* $\geq$ *maxRejects* **then**
      $O =$ **Mutation**(O)
    end
    $O =$ **LocalSearch**$(O, maxIter)$
    rejections = rejections + 1
  until ***AcceptOffspring****(Pop, O)*
  $I_R =$ **ReplacedIndiv**(Pop)
  $Pop = Pop - \{I_R\} + \{O\}$
end

colour class in the parent is ranked by calculating a score using the number of conflicts, the size of the class and the sum of the degrees of the vertices in the class. Lower scores indicate a better class. The ideal class has a lower number of conflicts with a larger number of vertices and greater sum of vertex degrees. Classes are ranked across all parents and the class with the lowest score is added to the offspring each time. Any vertices in the offspring that are already assigned a colour are ignored when calculating scores and adding new colour classes. After the crossover function produces an offspring, the Tabu Search algorithm is run on it to improve the colouring and eliminate any conflicts in a set number of iterations.

Algorithm 15: Well Informed Partition Crossover (WIPX)

input : parents $I_1, I_2, \ldots, I_n$
output: offspring O
O = empty, i.e. start with no vertex colour assigned
for *currentColor = 1* **to** k **do**
  foreach *parent* $I_i \in \{I_1, I_2, \ldots, I_n\}$ **do**
    foreach *colour class* I_i^c *in* I_i **do**
      1. Remove from I_i^c all vertices already coloured in O
      2. conflicts $= |\{(v_1, v_2) \in I_i^c \times I_i^c : (v_1, v_2) \in E\}|$
      3. classSize $= |I_i^c|$
      4. degreesCls $= \sum_{v \in I_i^c} \delta_v$
      5. score$[I_i^c] = conflicts - \frac{1}{|V|}(classSize + \frac{degreesCls}{|E| \times |V|})$
    end
  end
  set $(i^*, c^*) = argmin_{(i,c)} score[I_i^c]$
  foreach $v \in I_{i^*}^{c^*}$ **do**
    $O[v] = currentColor$
  end
end
foreach *unassigned* $v \in O$ **do**
  $O[v] = k$
end

AcceptOffspring: The offspring is evaluated for its distance from the population before it can be added. A target value of R is used as the minimum distance between the offspring and each population member for it to be accepted. The distance is calculated using the following equation: $d(I_A, I_B) = |V| - s(I_A, I_B)$. The function s calculates the similarity between two individuals in the population by calculating the maximum number of vertices in I_A that can remain in the same colour class when transforming I_A to I_B. If the offspring has a distance less than R from any population member it is rejected and repeatedly mutated until a diverse enough offspring is generated.

Mutation: The mutation function is used to maintain diversity among the population. This stops the

algorithm from converging too early and allows for a better solution to be found. It is applied to the offspring if it is rejected by the **AcceptOffspring** function more than a set number of times defined by $maxRejects$. It works by randomly changing the colours of a set number of vertices in the offspring. The number of vertices to mutate is increased each time the offspring is rejected. This can increase to the point that all vertices are mutated to produce a completely new colouring that is diverse from the population.

ReplacedIndiv: The ReplacedIndiv function is shown in Algorithm 16). The function first selects an individual from the population based on the fitness. The population member that has the least distance from the initial individual is then chosen, and the least fit of the two is selected for elimination. The offspring is added to the population in its place.

Algorithm 16: ReplacedIndiv replacement function

```
input  : population Pop = {I_1, I_2, ..., I_|Pop|}
output: the individual to be eliminated
repeat
  | C_1 = RandomIndividual(Pop)
until AcceptCandidate(C_1) (fitness-based acceptance)
minDist = maximum possible integer
foreach I ∈ Pop − {C_1} do
  | if d(I, C_1) < minDist then
  |   | if AcceptCandidate(I) then
  |   |   | minDist = d(I, C_1)
  |   |   | C_2 = I
  |   | end
  | end
end
if f(C_1) < f(C_2) then
  | return C_2
else
  | return C_1
end
```

2.7 Summary

As stated earlier, we wish to analyse the performance of both simple theory-driven algorithms and state-of-the-art evolutionary algorithms in our project. The behaviour of the $(1 + 1)$ EA, the $(2 + 2)$ GA with fitness sharing, randomised local search and both variants of iterated local search is quite well understood on simple graphs, but is relatively unknown when applied in more complex cases. Randomised local search behaves quite similarly to the $(1 + 1)$ EA, so testing the latter should be enough.

In contrast, the state-of-the-art algorithms such as the HCA, Evocol and Evo-Div have in the past been tested against each other on a variety of benchmark graphs, but theoretically they are poorly understood, and their performance has generally not been observed on more simple well-structured graphs to discover how they behave, so testing these against the more theory-driven algorithms should be of interest.

For this project, we intend to experiment using a subset of the above algorithms:

- $(1 + 1)$ EA
- HCA
- Evocol
- Evo-Div

This selection includes a simple theory based algorithm, the (1+1) EA, as a baseline for comparison, and the more modern approaches described: the Hybrid Colouring Algorithm, Evocol, and Evo-Div. Given the time-constraints of the project, this will also be a more manageable number of algorithms to investigate.

Chapter 3

Frameworks

A software framework is code which provides application-specific software by providing generic functionality which can be changed by additional user code. A framework provides reusable code which brings together the different required components needed for the development of the system. A framework manages the overall flow of control of the program and allows user extension of the framework by selectively overriding the required parts. Generally, the framework code is not meant to be modified but purely extended in order to make alterations.

3.1 DEAP

DEAP (Distributed Evolutionary Algorithms in Python) [9] is a framework designed to implement evolutionary algorithms for testing and prototyping. DEAP is designed to make the algorithms and data structures easily accessible and more transparent. The framework uses the Python programming language, which allows the implementation of the required parts of the evolutionary algorithm. The framework is designed to be lightweight, to focus on providing the general mechanisms and essential operators for implementing an evolutionary algorithm. This includes the implementation of a generic fitness function and a toolbox function. The toolbox function contains operators that will be used to implement an evolutionary algorithm; implementing these functions using a toolbox function allows for these operators to easily be changed once the algorithm has been written, without having to completely rewrite the algorithm.

As DEAP was developed for testing evolutionary algorithms, it contains some tools to help interpret the results of the algorithm. The first of these is a Best-of-Run function. This allows the best individual at each evolution to be stored, which allows the path through the evolutions to easily be recreated after the algorithm is complete. The next tool is a checkpointing tool, which allows for a snapshot of the evolution of the algorithm to be taken. This means that the evolutions can be restarted midway through, allowing changes to made to the algorithm specifically to compare a set phase of the evolutions. DEAP also contains a tool for easily implementing parallelisation, it contains a Distributed Task Manager to allow tasks to be split into sub-tasks and run in parallel. This allows algorithms to be run more efficiently, which is important when testing algorithms.

3.2 Open BEAGLE

Open BEAGLE [10] is a powerful evolutionary algorithm framework implemented in C++. The framework aims to be efficient, which is why the programming language was chosen and critical aspects of the code are optimised. Open BEAGLE provides the core aspects for implementing an evolutionary algorithm using an object oriented approach to allow the reuse of key aspects of any evolutionary algorithm. The framework has several core aims, including user friendliness and robustness. The aim to be user friendly means the framework provides an easy-to-use programming interface. The framework contains validation within the code which ensure that the implementation is correct, ensuring the robustness of the code. The programming interface of the framework provides generic functions to implement populations, evolutions and a set of evolutionary algorithm operators. These generic functions can all be specialised in order to alter the implementation. These functions include generic fitness and evolutionary models that can be extended. Open BEAGLE implements data input and output using XML files, allowing the output of the algorithm to be easily interpreted.

3.3 Framework Comparison

	DEAP	Open BEAGLE
Language	Python	C++
Framework type	White Box	Black Box
Generic Fitness	Yes	Yes
Generic Operations	Yes	Yes
Generic Evolutionary Models	Yes	Yes
Checkpointing Features	Yes	No
Parameter Management	Yes	Yes
Configurable Outputs	Yes	Yes
Distribution via Parallelism	Yes	No

3.4 Framework Choice

While both DEAP and Open BEAGLE both have very similar features, DEAP has some clear advantages that mean it is the best framework to use for our implementation. The first is the checkpointing features implemented within DEAP, which allows the saving of the state of the evolutions. This means that the algorithm can be restarted and tested at multiple points within the evolution. The next is the wide array of control offered by DEAP, as it is a white box framework which means that the data types and algorithms are transparent. As DEAP incorporates easy parallelisation, this means that algorithms can be run more efficiently. This is an advantage when testing algorithms as it allows changes to be made and tested quickly. The framework also contains good benchmarking tools with configurable outputs, meaning that the data gathered for comparisons can easily be controlled.

3.5 Automation of running implemented algorithms

Using Python and the DEAP framework allows for automation of running the tests and algorithms that are implemented. This allows all tests to easily be reviewed and rerun in order to review any parameters or results. This can be done using a Python script to set the parameters of each test, or by using a shell script to run the algorithm with set parameters.

Chapter 4

Implementation

In this chapter, we will discuss the implementation of the different algorithms using Python and the DEAP framework, and also the implementation of the different graphs that we used in testing.

4.1 Representing the Graphs

To implement the graphs for testing, we made use of the Python package NetworkX [11]. NetworkX provides lots of tools that are useful for when working with graphs, including methods to draw graphs with a specified colouring.

In the package grpahs are stored as a dictionary of dictionaries of dictionaries, which allows for fast lookup and reasonable memory storage even for large, sparse graphs. The first level of keys allows for looking up adjacent nodes in the graph, i.e for a graph G with node u, looking up $G[u]$ will return an adjacency dictionary for node u. The second level of keys returns any edge attributes, i.e. for a pair of nodes (u, v) joined by an edge, looking up $G[u][v]$ returns the attributes of the edge(s) that joins them. This kind of implementation is beneficial compared to list storage as it allows for fast look up of graph attributes. It is also preferable to sets as it allows for multiple edges between nodes, and allows for attributes to be stored for the edges..

4.2 (1+1) EA

DEAP contains a built in function for implementing a (1+1) EA. The function *deap.algorithms.eaMuPlusLambda* is an implementation of the $(\mu + \lambda)$ evolutionary algorithm, and by setting $\mu = \lambda = 1$ the (1+1) EA can be obtained.

The function takes as parameters a list of individual (the population), the DEAP toolbox with specified evolution operators (like the crossover and mutation functions), the values for μ and λ, the probability of crossover and mutation, the number of generations, and some optional parameters for storing the best individuals.

4.3 Hybrid Colouring Algorithm

The hybrid colouring algorithm is described by Algorithm 8 in Section 2.4. It is composed of 5 main sections: the population initialisation function, the select parents function, the crossover function, the tabu search mutation function, and the population update function. The implementation of these separate components is described below.

InitPopulation: For each individual, the function initialises an individual with each node having a value of $k+1$, where k is the number of colours, using a 1D array. The function also generates k empty colour classes, represented as a 2D array, which will contain an array of individuals belonging to that class. For each vertex in the individual, the number of viable classes is calculated by looking at the neighbours of each vertex. Then the vertex is chosen that has the minimum number of viable classes, and it is assigned it's minimum colour class. Once all vertices have been assigned their minimum viable colour class, any vertices that have not been assigned a colour class due to conflicts, it will still have a value of $k + 1$, and so is allocated to a random colour class. This must be done after all vertices have been iterated over once, as a random assignment of a vertex may cause conflicts with a vertex that would otherwise be assigned to that class.

Due to the greedy nature of this step, the outcome was highly affected by the representation of the graph. The algorithm for this step would assign nodes in the order they were added to the graph, and so the same graph could be given a different colouring depending on the order it was constructed using the Networkx package. To

keep this consistant between graphs, nodes were added to the graph in the order they appeared in the graphs edges, starting with the edge belonging to the smallest indexed nodes as they were indexed in the DIMACS set.

SelectParents: This is implemented using a standard random selection function from the DEAP toolbox, where a specified number of parents are chosen from the population. In this case, the number of parents selected is 2.

Crossover: The offspring is initialised with all vertices having a value of -1. This is useful to see when vertices have been assigned a class and which vertices remain unassigned. Then copies of each parent are produced, so that vertices can be deleted. The algorithm then alternates between parents whilst there are still nodes yet to be reassigned in each parent, and chooses the largest colour class (colour class containing the most vertices) by converting the parent to a set and then taking the size of each set. The indices of the vertices belonging to that class are then stored in an array. This array is then stored in a 2D colour class array (similar as to in the **InitPopulation** function). When each colour class has been assigned vertices, the offspring is assigned colours to each vertex. Any unassigned vertices are allocated a random colour class.

TabuSearch: The tabu search function searches for local improvements to the individual, avoiding moves that have been recently tested. To do this it uses a tabu list, which in this case is implemented using a 2D array. The function stores the best found individual, and compares new individuals to this and replaces it if there is an improvement. The tabu search first gets a list of all conflicts contained in the individual. Within the list of conflicts it then searches for a best move, a colour change yields the biggest reduction in the number of conflicts. It then compares the number of conflicts in the new individual to the current best individual, and if an improvement is made then the best individual is replaced. This process is repeated for a set number of iterations. The tabu tenure (the amount of time a move is considered tabu) is set using two parameters A and α in the function :

$$tabu\_tenure = random\_int(0, A) + \alpha \times num\_conflicts \tag{4.1}$$

where *random_int* returns an integer in the range $[0, A-1]$. In [6], Galinier & Hao use the parameter values of $A = 10$, $\alpha = 0.6$, and so these are the values used in our implementation.

UpdatePopulation: To update the population, the parent with the worse fitness (higher number of violations) is replaced by the offspring. This occurs regardless of whether the offspring has fewer conflicts than the worse parent or not.

4.4 Evocol

Evocol is described by Algorithm 11. Similarly to HCA, it is composed of 5 main sections: the population initialisation function, the select parents function, the crossover function, the tabu search mutation function, and the population update function. The implementation of these separate components is described below.

InitPopulation: Initialising the population for Evocol is much simpler than it is in HCA. To initialise an individual, we simply start with an individual where all the vertices have colour 0, and randomly assign a colour to each vertex. We then repeat this process for the number of individuals in the population - we use 15, as this is the value used in the Evocol paper.

SelectParents: This is implemented using a standard random selection function from the DEAP toolbox, where a specified number of parents are chosen from the population. In this case, the number of parents selected is 3.

Crossover: The multi-parent crossover operation proceeds as follows. First we initialise the offspring, starting with each vertex labelled as -1. Then, make copies of the parents and initialise a dictionary to store scores for each pair of parent and colour class. Now we loop over the possible colours from 1 to k. First, for each parent, we remove the colours already assigned (not equal to -1), calculate the number of possible conflicting pairs by finding the length of a list comprehension, then calculate the score using the simple formula given in the algorithm. After doing this for each parent, we find the minimum score in the dictionary, and assign the relevant vertices in the offspring to the colour of that colour class. After repeating this for each colour, we then loop over the vertices in the offspring, and for remaining ones that have not yet been assigned, we simply check the cost of each possible colour and then assign each one to the best choice.

TabuSearch: Tabu search proceeds in much the same way as it does in the HCA; the only difference is in how the tabu tenure is calculated. The tabu tenure (the amount of time a move is considered tabu) is set using three parameters A, α and M_{max} in the function:

$$tabu\_tenure = random\_int(0, A) + \alpha \times num\_conflicts + \lfloor \frac{M}{M_{max}} \rfloor \tag{4.2}$$

where *random_int* returns an integer in the range $[1, A]$ and M is the number of the last consecutive moves that kept the fitness function constant. In [6], Evocol uses the parameter values of $A = 10$, $\alpha = 0.6$, and $M_{max} = 1000$, and so these are the values used in our implementation.

UpdatePopulation: Updating the population is a more involved process than in the HCA, as Evocol aims to control population diversity as well as fitness. First, we repeatedly select random candidates from the population until one is accepted. (The process of accepting a candidate is a simple probabilistic formula described in the algorithm.) Once one has been accepted, this is designated as our first possibility for elimination. We then choose a second possible candidate, by calculating the distance between the first candidate and each other individual in the population, and then selecting the closest one which is also accepted for elimination. (Distance is also calculated by a simple mathematical formula described earlier in the algorithm.) The candidate with the lowest number of conflicts from these two options is eliminated.

4.5 Evo-Div

Selection: The parents are selected randomly from the population using the *deap.tools.selRandom* function available in DEAP.

Crossover: The crossover function takes a list of n parents and produces an offspring colouring from the parent colourings. A 1D array is initialised with -1 values to represent the offspring. The best colour classes are selected from the parents by evaluating each colour class in each parent. The parents are each represented by lists where each element is a colour and its index is a node. This representation is converted to a dictionary of colour classes where each key in the dictionary is a colour and the value is a set of indices representing the nodes that are that colour in the individual. This allows the class conflicts and class degrees to be easily computed for each colour class. The scores are calculated and stored in a dictionary where the key is a score and the value is a tuple (i, c) where i is the parent and c is the colour. The colours are looped through in a for loop and the class with the minimum score is selected. All nodes from the class are assigned the current colour in the offspring. At the end of the loop if any nodes remain uncoloured, they are assigned the last colour.

Mutation: The mutation function randomly changes the colour of a set number of nodes in the individual. This was implemented by generating a list of indices for the number of nodes using the range function in python. The list is then randomly shuffled using *random.shuffle* and the first R elements are chosen. The colours at these R nodes are changed by randomly assigning new colours using *random.randint*.

Tabu Search: The tabu search algorithm was implemented using the pseudo code and data structures described in [12]. A 2D array M was initialised with size $(k \times n)$ where k is the number of colours and n is the number of nodes for the solution matrix. Each value $M[i, j]$ in the array is filled with the number of conflicts that would be at vertex j if it had the colour i. The best move is found by looping through the solution matrix and finding the maximum change in the number of conflicts. This is achieved by calculating the difference between each value at $M[c(i), i]$ and $M[j, i]$ where i is a node, $c(i)$ is the current colour of the node and j is a colour where $j \neq c(i)$. The colour and vertex with the best move are also stored and applied once the best move is found and the values in the solution matrix updated. The move is stored in the tabu map and tabu queue. The tabu map was implemented as a 2D array of size $(k \times n)$ initialised with zeros and the tabu queue as a double ended queue *collections.deque* from the collections module. Once a move $(colour, vertex)$ is applied, the value at that move is updated to 1 in the tabu map and added to the tabu queue which means that the move is not allowed. Elements are popped from the left of the queue until the length of the tabu queue is below the tabu tenure T_l and the value in the tabu map of the popped moves are changed back to zero to represent that the moves are legal once again.

$$tabu\_tenure = \alpha \times f(c) + random\_int(0, A) + \lfloor \frac{L}{L_{max}} \rfloor \tag{4.3}$$

Equation 4.3 shows how the tabu tenure T_l is calculated where L is the number of iterations the fitness remains unchanged and L_{max} is the maximum number of iterations where the fitness is allowed to remain unchanged. The values used are $\alpha = 0.6$, $A = 10$, $L_{max} = 10$.

Replacement: The replacement function selects the two individuals from the population that are closest in distance and selects the individual with the higher fitness value for elimination. The distance metric is computed by calculating the difference between the number of nodes in the graph and the similarity between two individuals. The similarity is computed using *scipy.optimize.linear_sum_assignment* which computes the minimum changes in vertex colours required to change one individual to another.

Chapter 5

Testing & Results

5.1 Testing

We tested the various evolutionary algorithms using selection graphs from a large collection for benchmarking colouring algorithms introduced by the second DIMACS Implementation Challenge [13], a collection which has been extensively used in the literature for many years.

To test our graphs we chose 3 different graphs from the DIMACS graph test set. The graphs chosen were:

- **DSJC125.1.col**: a random graph with 125 nodes and 736 edges.
- **queen5_5.col**: a queen graph *(see below)* with 25 nodes and 160 edges.
- **myciel5.col**: a graph based on the Mycielski transformation *(see below)* with 47 nodes and 236 edges. These graphs are difficult to solve because they contain no triangles.
- **mulsol.i.3.col**: a graph with 184 nodes and 3916 edges, and is based on register allocation for variables in real codes.

A queen graph is a graph based on an $n \times n$ square chessboard, where each node represents a square. Two nodes are connected by an edge if they are in the same row, column or diagonal on the chessboard.

The Mycielski transformation [14] is a method of constructing graphs from smaller triangle-free graphs, preserving their triangle free nature but increases the chromatic number. Applying the transformation multiple times allow for arbitrary chromatic number to be reached. Applying the transformation to a graph G with n vertices and m edges, the transformed graph $Mycielski(G)$ has $2n + 1$ vertices and $3m + n$ edges.

These graphs were chosen to be representative of some complex graphs that can be encountered in theoretical and real situations.

We also chose to test the various algorithms on more simple types of graphs, for which theoretical results have already been shown for basic algorithms. This will be interesting in that it will allow us to see whether more complex algorithms can provide even better performance in these well-understood cases, or whether the increased overhead involved in running a more involved algorithm outweighs any possible benefits. These graphs included:

- **10 node cycle graph**: chromatic number = 2
- **11 node cycle graph**: chromatic number = 3
- **15 node binary tree graph**: chromatic number = 2
- **11 node wheel graph**: chromatic number = 3
- **12 node wheel graph**: chromatic number = 4
- **Petersen graph**: chromatic number = 3

A cycle graph is a graph where the i-th vertex is neighboured to vertices $i-1$ and $i+1$ (identifying vertex 0 with vertex n). It has been shown [3] that the expected optimisation time for the simple $(1 + 1)$ EA algorithm on a cycle graph is $O(n^3)$. A complete binary tree is a graph where every node either has two children or is a leaf, and all the leaves are on one level. To ensure the vertices of a subtree form a coherent sequence, we encode the tree by enumerating the vertices with an in order traversal. Similarly, we know [15] that the $(1 + 1)$ EA requires an expected time of at least $2^{\Omega(n)}$ to find an optimum. It is worth nothing that these theoretical results were shown while investigating a different problem known as the Ising model, this model can be seen [15] as an

Table 5.1: Chromatic numbers for the simple test graph

Graph	**Chromatic Number**
10 Cycle Graph	2
11 Cycle Graph	3
11 Wheel Graph	3
12 Wheel Graph	4
15 Binary Tree Graph	2
Petersen Graph	3

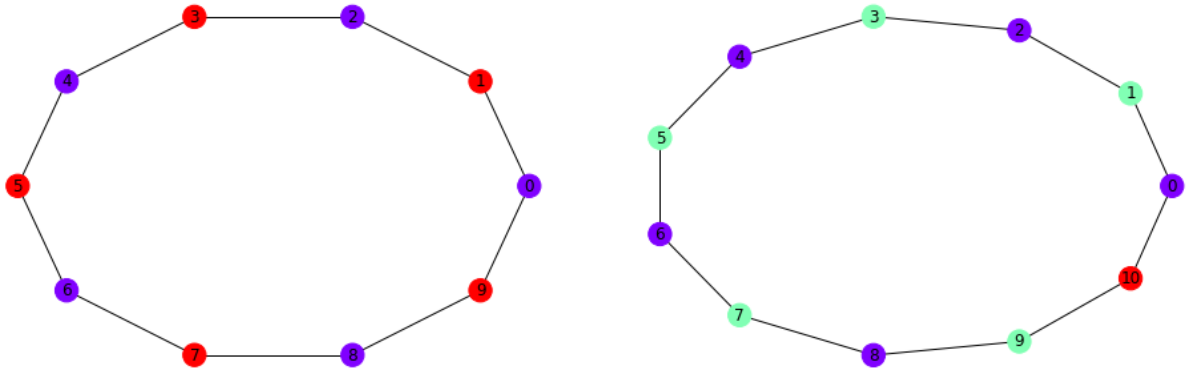
Table 5.2: Chromatic numbers for the DIMACS test graph

Graph	**Chromatic Number**
DSJC.125.1	5
queen5_5	5
myciel5	6
mulsol.i.3	31

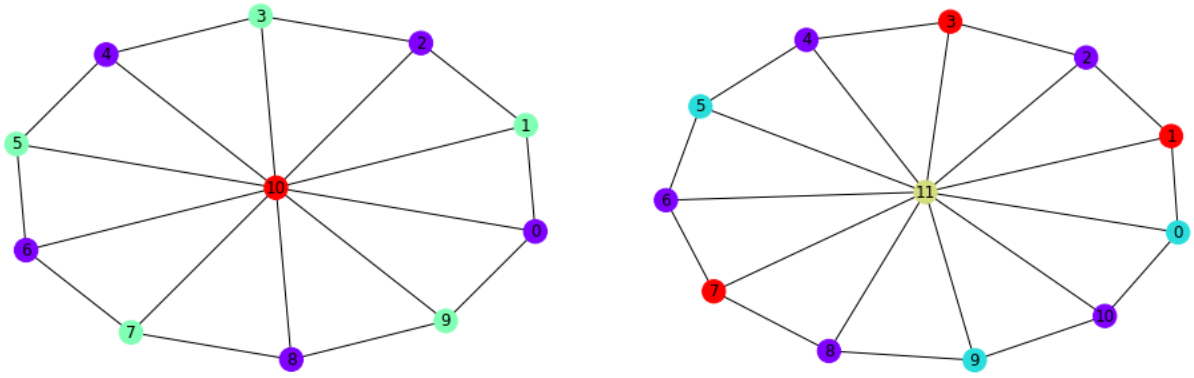
inverse graph colouring problem since the Ising model rewards edges between vertices of the same colour and the graph colouring problem rewards edges between vertices of differing colour, so the results still hold in the case we are interested in.

These graphs were chosen as they are all easy to draw and be presented. Choosing a simple graph or arbitrary size doesn't necessarily make it more complex to solve, but ease of presentation of the graphs is a useful property. These simple graphs can be seen with their optimum colouring in Figure 5.1.

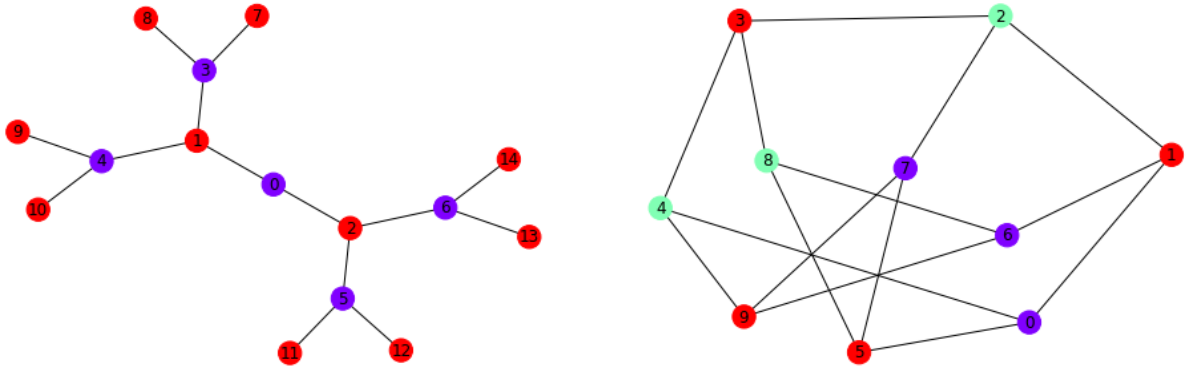
By testing the algorithms on these graphs, we will be able to test both the simple and the state-of-the-art algorithms in a variety of different scenarios, as experimentation will give us a guide to how the algorithm realistically performs in a way that would not be achievable by simply analysing the algorithms theoretically.



(a) A 10 node cycle graph *(left)* and 11 node cycle graph *(right)*



(b) An 11 node wheel graph *(left)* and 12 node wheel graph *(right)*



(c) A 15 node binary tree *(left)* and Petersen graph *(right)*

Figure 5.1: Simple test graphs, all with their optimum colouring

5.2 Results

To look at the performance of the algorithms we will look at how they each perform over 100000 evaluations, averaged over 30 runs, when searching for a solution using the optimum number of colours.

We will be looking at the fitness achieved per number of evaluations to analyse the performance over time of the algorithms, as well as looking at the average final fitness of each algorithm. Here, the fitness function is defined as the number of conflicts in the colouring. In cases where algorithms have a population of individual with size > 1 the best fitness will be considered. Looking at the number of fitness evaluations is a commonly used metric in the field, as it is machine independent and is unbiased towards different complexity algorithms.

It is also useful to look at the correct allocation percentage of the algorithms final fitness, calculated as:

$$\frac{(total\_edges - final\_conflicts)}{total\_edges} \times 100\% \tag{5.1}$$

This metric will allow for useful comparisons to be drawn for results on graphs of different sizes.

As well as looking at the number of fitness evaluations required to reach a valid k-colouring, we will also look at the success rate for each graph where a successful run is one which the algorithm finds a proper k-colouring for the graph.

$$\frac{successful\_runs}{total\_runs} \times 100\% \tag{5.2}$$

For the results presented we considered the success rates after 100000 evaluations. This is useful as the algorithms each include some form of randomness to optimise the colouring found.

In the following sections, each algorithms performance on the graphs will be discussed, followed by a comparison of the algorithms on the different graphs.

5.2.1 Simple Graphs

The chromatic numbers for the simple test graphs are presented in Table 5.1.

(1+1) Evolutionary Algorithm

To test the (1+1) Evolutionary Algorithm on these simple graphs, we ran it for 100000 generations with a mutation probability of $\frac{1}{n}$ where n is the number of nodes in the graph. The results can be seen in Figure 5.3. The algorithm could achieve an optimal solution for each graph, but performed worse on some graphs than others on average. The algorithm performed best on the 11 cycle graph, 12 wheel and Petersen graph, achieving an average final conflicts of 0, 0.4 and 0.4 respectively. The algorithm performed less well on the 10 cycle graph, 11 wheel graph and 15 binary tree graph achieving an average final conflicts of 2, 1.6 and 1.8 respectively.

From these results it appears that whilst the (1+1) EA performs well on graphs with odd length cycles, it has a lower performance for graphs containing even length cycles or no cycles. This is likely due to the fact that for these chosen graphs, the even length cycles have a unique colouring, as the colours in the cycle must alternate between the two colours. However, for the odd length cycles there are many options for a solution. Also, the graphs with odd cycles have a higher chromatic number, and so it would be expected that it would be easier to find an optimal colouring as the algorithm has more colour classes to choose from.

Graph	Success Rates (%)	Average Final Fitness
10 Cycle Graph	50.0	0.5
11 Cycle Graph	100.0	0.0
11 Wheel Graph	70.0	0.3
12 Wheel Graph	100.0	0.0
15 Binary Tree Graph	26.7	0.9
Petersen Graph	100.0	0.0

Table 5.3: Success Rates for (1 + 1) EA on simple graphs

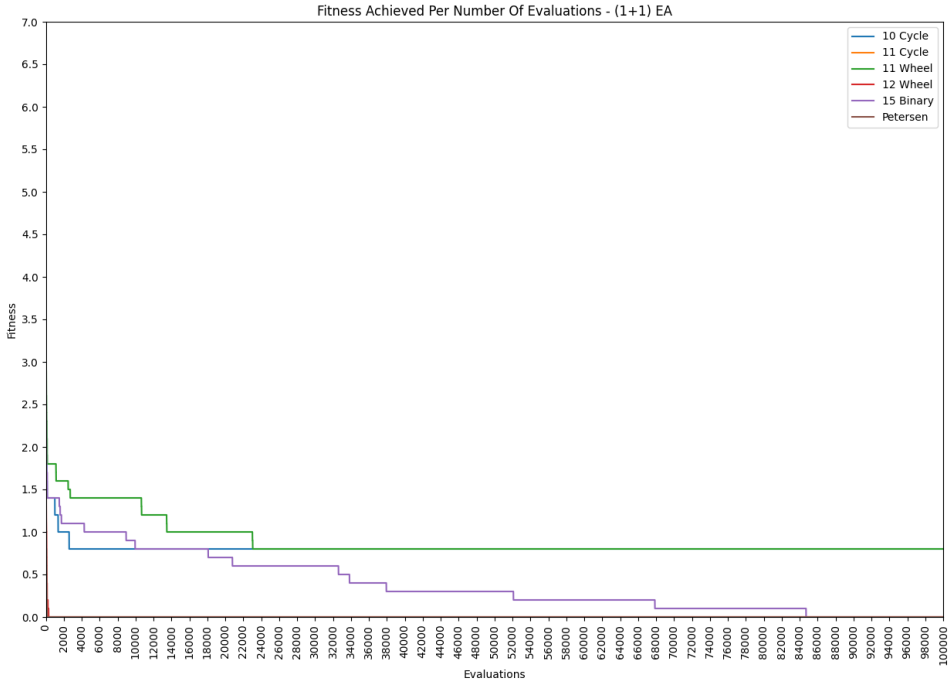


Figure 5.2: Average fitness achieved for each generation of (1+1) EA on simple test graphs

Hybrid Colouring Algorithm

Due to the nature of the initialisation function in the Hybrid Colouring Algorithm, an optimum colouring on 10 cycle graph, 11 cycle graph and 15 binary tree can be achieved immediately. For these three graphs the greedy saturation algorithm used for initialisation is sufficient to find a solution.

For the remaining three graphs, the HCA could find an optimal colouring with a small number of iterations. The parameter values for the following results were found with a tabu search depth of 1000 and a population size of 10. The success rates of the algorithm on the 11 wheel, 12 wheel and Petersen graphs are shown below.

Graph	Success Rates (%)	Average Final Fitness
11 Wheel Graph	100.0	0.0
12 Wheel Graph	100.0	0.0
Petersen Graph	100.0	0.0

Table 5.4: Success Rates for HCA on simple graphs

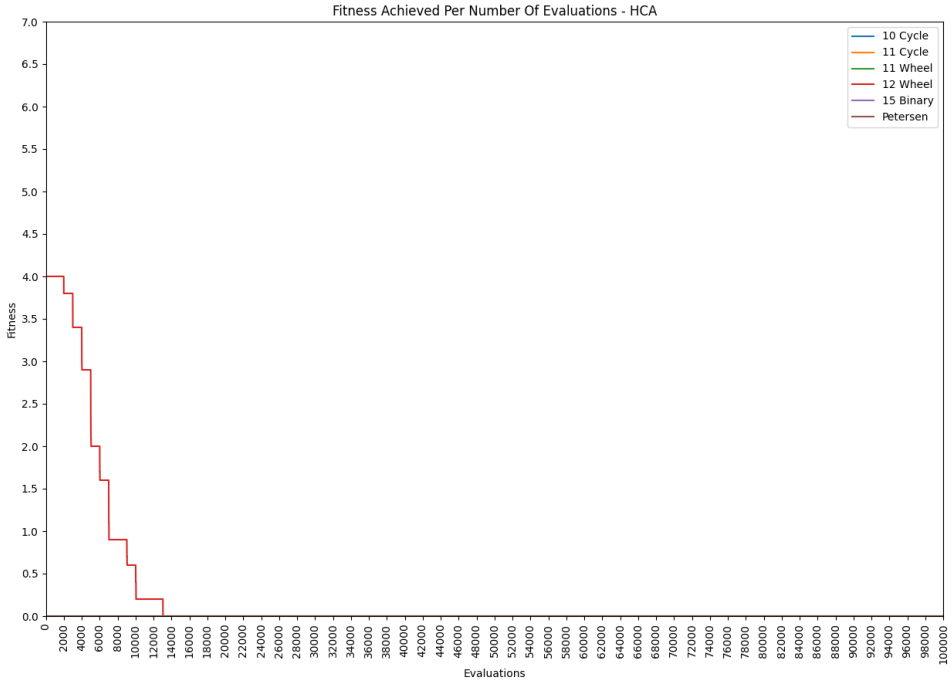


Figure 5.3: Average fitness achieved for each generation of HCA on simple test graphs

Evocol

Evocol can find colourings for all of the small simple graphs in a relatively small number of evaluations. On larger graphs with a simple structure, it can sometimes struggle to find a solution; this is most likely due to the diversity control measures, as for graphs without much structure it is hard for offspring to differ substantially from each other, and the algorithm can sometimes end up in loops of trying to find more diverse solutions. The following results were found with a tabu search depth of 1000 and a population size of 15.

Graph	Success Rates (%)	Average Final Fitness
10 Cycle Graph	100.0	0.0
11 Cycle Graph	100.0	0.0
11 Wheel Graph	100.0	0.0
12 Wheel Graph	100.0	0.0
15 Binary Tree Graph	80.0	0.2
Petersen Graph	100.0	0.0

Table 5.5: Success Rates for Evocol on simple graphs

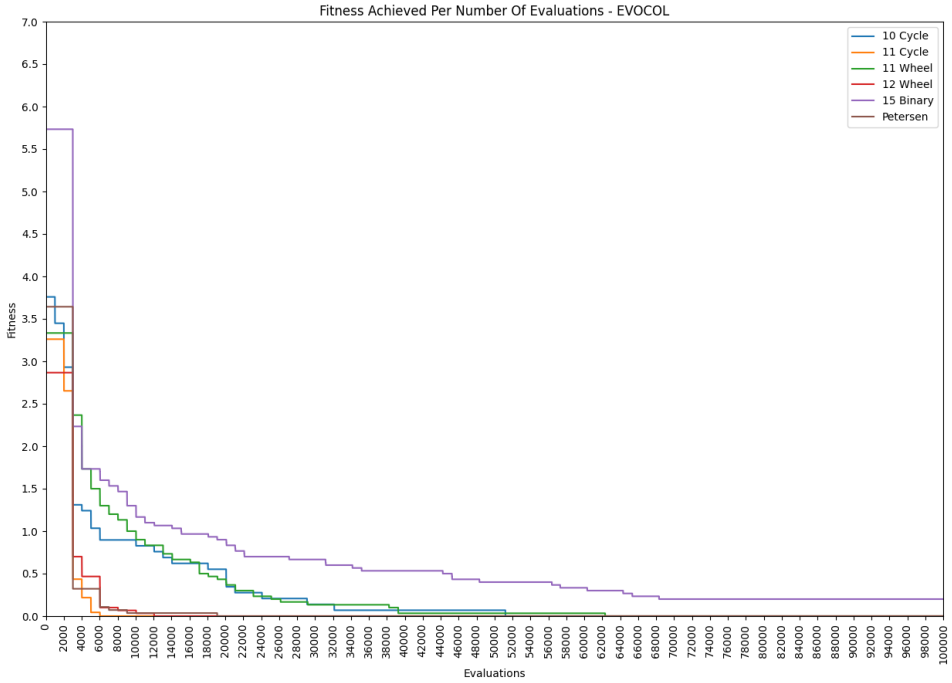


Figure 5.4: Average fitness achieved for each generation of Evocol on simple test graphs

Evo-div

Evo-div is able to find colourings for all runs on all simple graphs except the 15 Binary Tree graph where it did not manage to find an optimal colouring in 100,000 evaluations in a few of the runs. It is able to find a colouring very quickly for the 11 cycle, 12 wheel and Petersen graphs, but takes thousands of evaluations to find colourings for the 10 cycle, 11 wheel and 15 binary tree graph.

Graph	Success Rates (%)	Average Final Fitness
10 Cycle Graph	100.0	0.0
11 Cycle Graph	100.0	0.0
11 Wheel Graph	100.0	0.0
12 Wheel Graph	100.0	0.0
15 Binary Tree Graph	90.0	0.1
Petersen Graph	100.0	0.0

Table 5.6: Success Rates for Evo-div on simple graphs

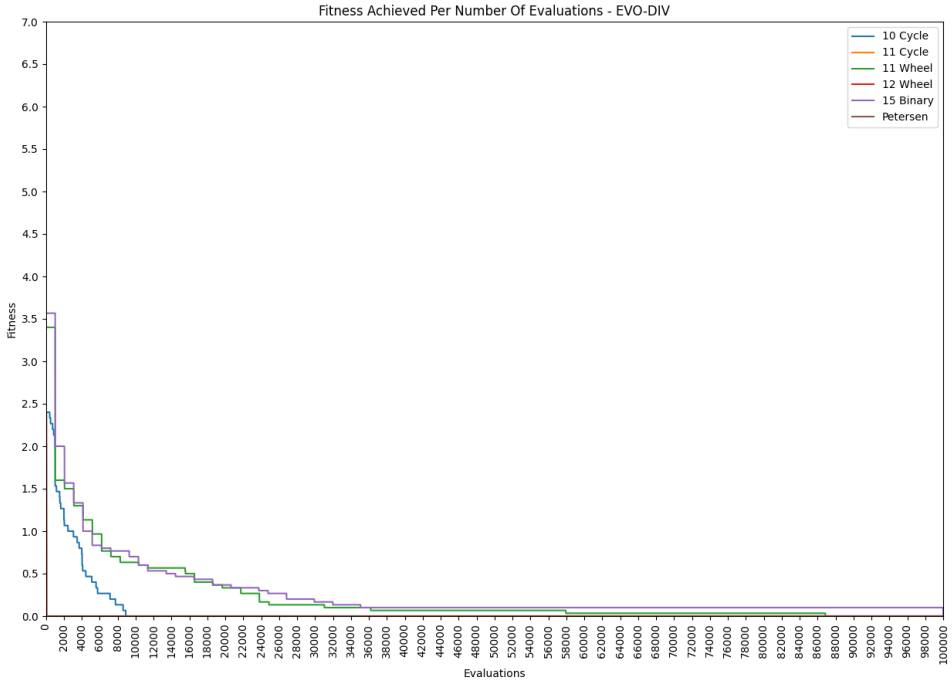


Figure 5.5: Average fitness achieved for each generation of Evo-div on simple test graphs

Graphs Comparison

A comparison of the performance of the different algorithms for each of the simple graphs can be seen in Figure 5.6[1]. As can be seen from the graphs, the HCA shows the best performance on each of the simple graphs, being the fastest to reach an optimal solution for each graph we chose to test.

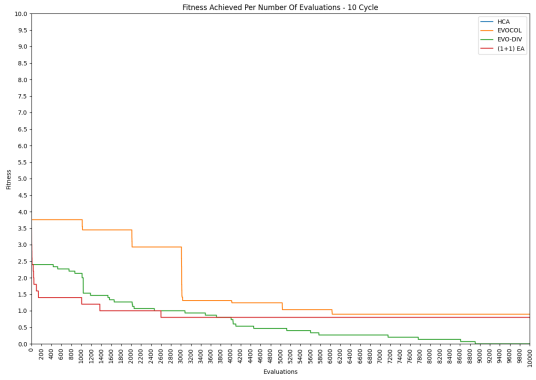
Looking at the remaining three algorithms, for a small number of initialisations the (1+1) EA outperforms both Evocol and Evo-Div. However, given a larger number of evaluations the two complex algorithms catch up with the fitness achieved by the (1+1) EA and also achieved a lower final fitness.

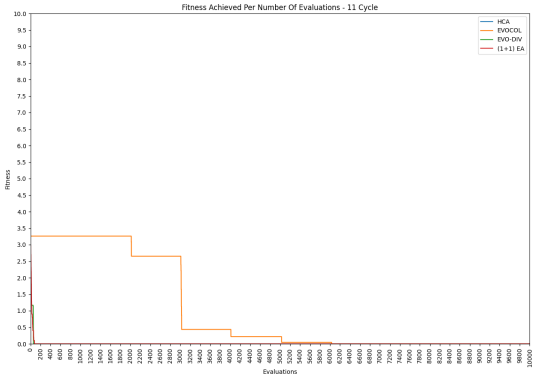
The success rates of the algorithms on the simple graphs after 10000 evaluations are shown in the table below.

Graph	(1+1) EA	HCA	Evocol	Evo-div
10 Cycle Graph	50.0	100.0	100.0	100.0
11 Cycle Graph	100.0	100.0	100.0	100.0
11 Wheel Graph	70.0	100.0	100.0	100.0
12 Wheel Graph	100.0	100.0	100.0	100.0
15 Binary Tree Graph	26.7	100.0	80.0	90.0
Petersen Graph	100.0	100.0	100.0	100.0

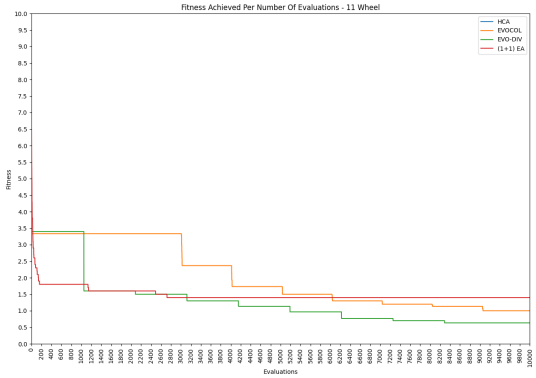
Table 5.7: Success Rates for all algorithms on simple graphs

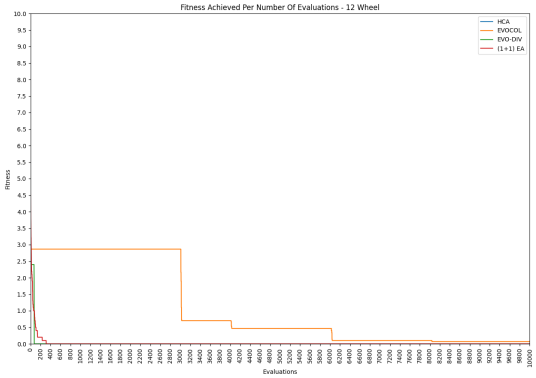
[1]Only data for the first 10000 evaluations is shown, rather than the full 100000 evaluations.



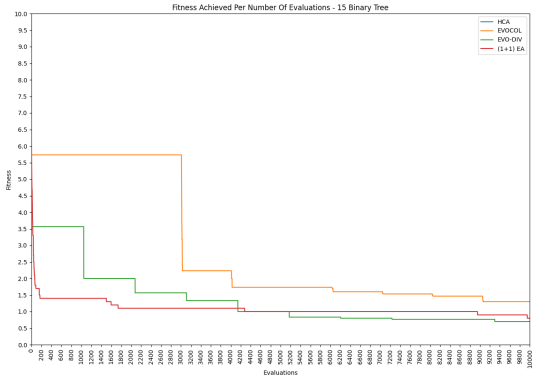


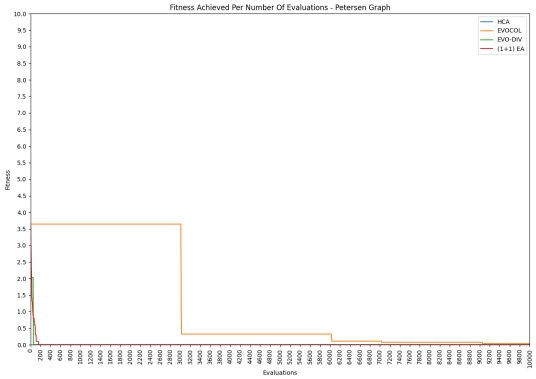
(a) 10 node cycle graph *(left)* and 11 node cycle graph *(right)*





(b) 11 node wheel graph *(left)* and 12 node wheel graph *(right)*





(c) 15 node binary tree *(left)* and Petersen graph *(right)*

Figure 5.6: Comparison of the performance for each of the algorithms on the simple graphs

5.2.2 DIMACS Graphs

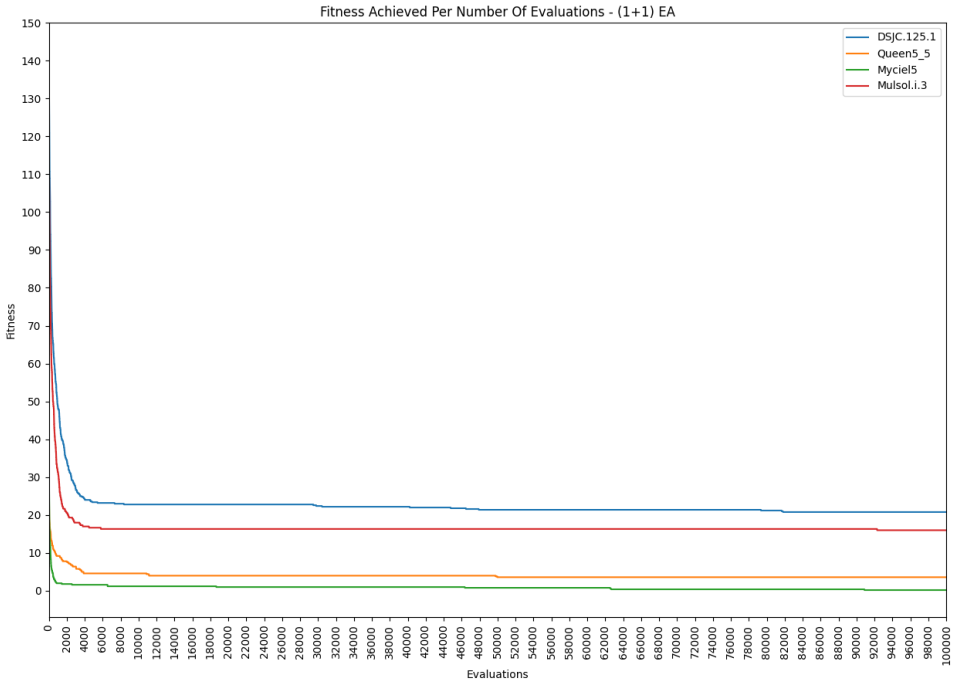
The chromatic numbers for the chosen DIMACS graphs are presented in Table 5.2[2].

(1+1) Evolutionary Algorithm

To test the (1+1) Evolutionary Algorithm on the DIMACS graphs we ran it with the same configuration as for the simple graphs, with a mutation probability of $\frac{1}{n}$ where n is the number of nodes in the graph. The results can be seen in Figure 5.7.

As shown in the results, the (1+1) EA performs better on the smaller graphs, Myciel5 and Queen5_5 with a lower performance on the larger graphs. The algorithm makes a very quick improvement for a smaller number of evaluations before reaching a plateau between 1000 and 5000 evaluations depending on the graph.

Due to the (1+1) EA only being able to make small jumps between colour configurations, it is very likely to get stuck in locally optimum configurations which it can then not escape to reach the solution. The (1+1) EA was never able to reach a solution for any of the DIMACS graphs we chose to test the algorithms on.



g

Figure 5.7: Average fitness achieved for each generation of (1+1) EA on DIMACS test graphs

Hybrid Colouring Algorithm

The hybrid colouring algorithm initially performs very well on the DIMACS graphs, and is able to make large improvements in fitness in a small number of iterations. However, on all the graphs, excluding Mulsol.i.3, the algorithm quickly reaches a plateau where the same improvements are no longer seen. For Mulsol.i.3, the HCA manages to maintain a much steadier rate of improvement for much longer before reaching a plateau.

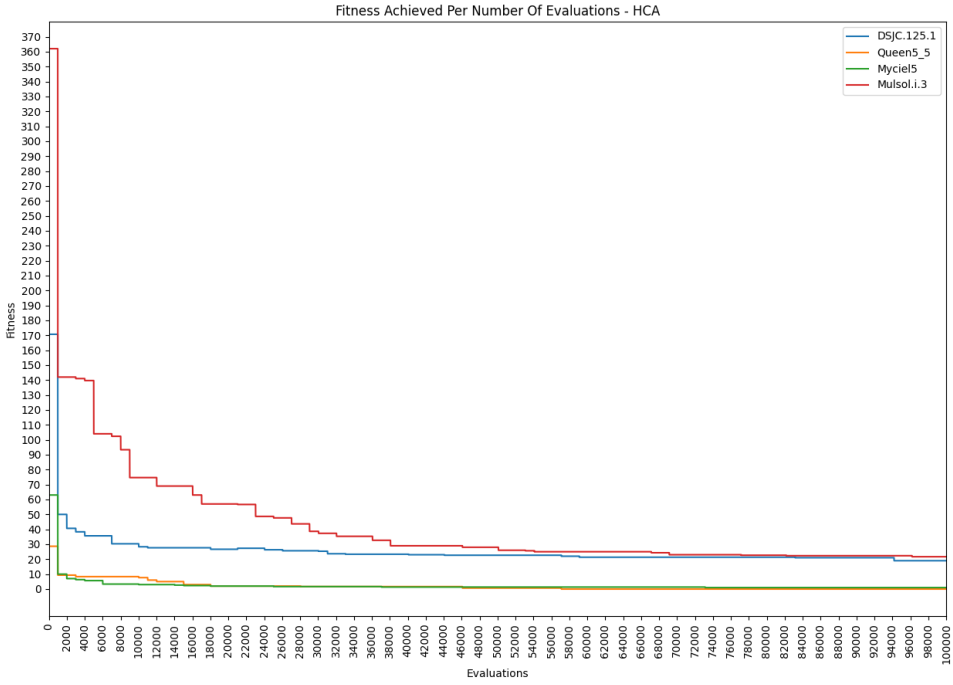
One reason the HCA quickly reaches this plateau may be down to its initialisation function. Whilst it potentially is able to achieve a higher fitness than through random initialisation, it could also reduce the diversity of the population by making all the individuals much more similar.

The results of the HCA on the different DIMACS graphs can be seen in Figure 5.2.2.

[2]The chromatic number presented for DSJC.125.1 is the best upper bound for the graphs chromatic number

Graph	Success Rates (%)	Average Final Fitness
DSJC.125.1	0.0	20.7
Queen5_5	100.0	0.0
Myciel5	36.7	0.6
Mulsol.i.3	0.0	21.7

Table 5.8: Success Rates for HCA on DIMACS graphs



g

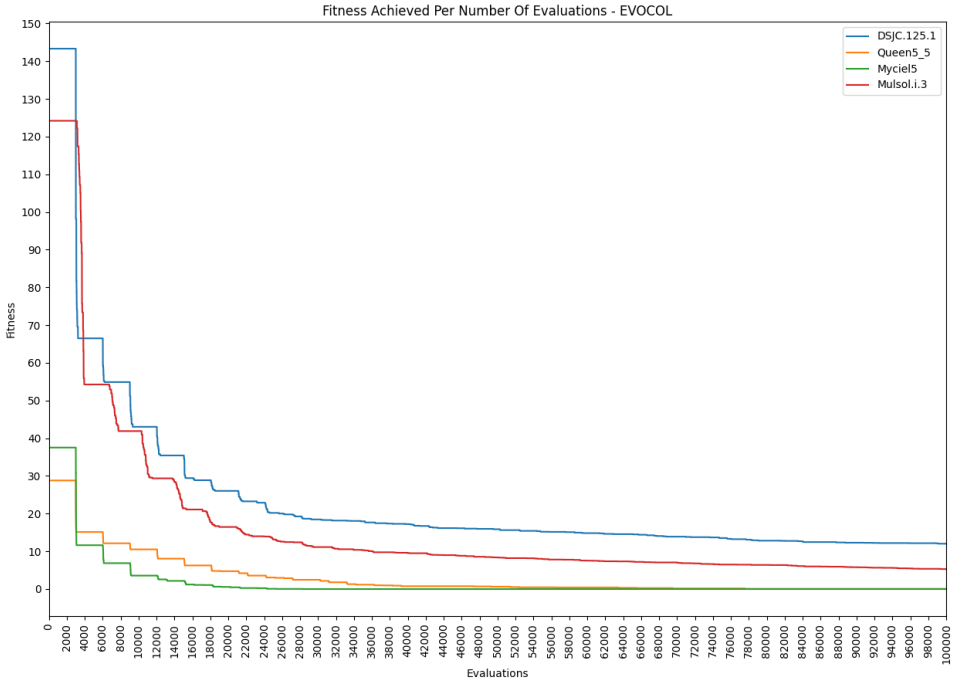
Figure 5.8: Average fitness achieved for each generation of HCA on DIMACS test graphs

Evocol

Evocol generally performs well on the DIMACS graphs, although it was only able to find the optimal colouring on all runs for the Myciel5 graph within the 100,000 evaluations. For the Queen5 graph evocol managed to find the optimal solution 96.7% of the time, showing that it was successful during most runs. On all graphs evocol was unable to make any significant improvements to the fitness after 30,000 iterations, showing that more iterations would in most cases not improvement the final fitness.

Graph	Success Rates (%)	Average Final Fitness
DSJC.125.1	0.0	12.0
Queen5_5	96.7	0.07
Myciel5	100.0	0.0
Mulsol.i.3	0.0	5.3

Table 5.9: Success Rates for Evocol on DIMACS graphs



g

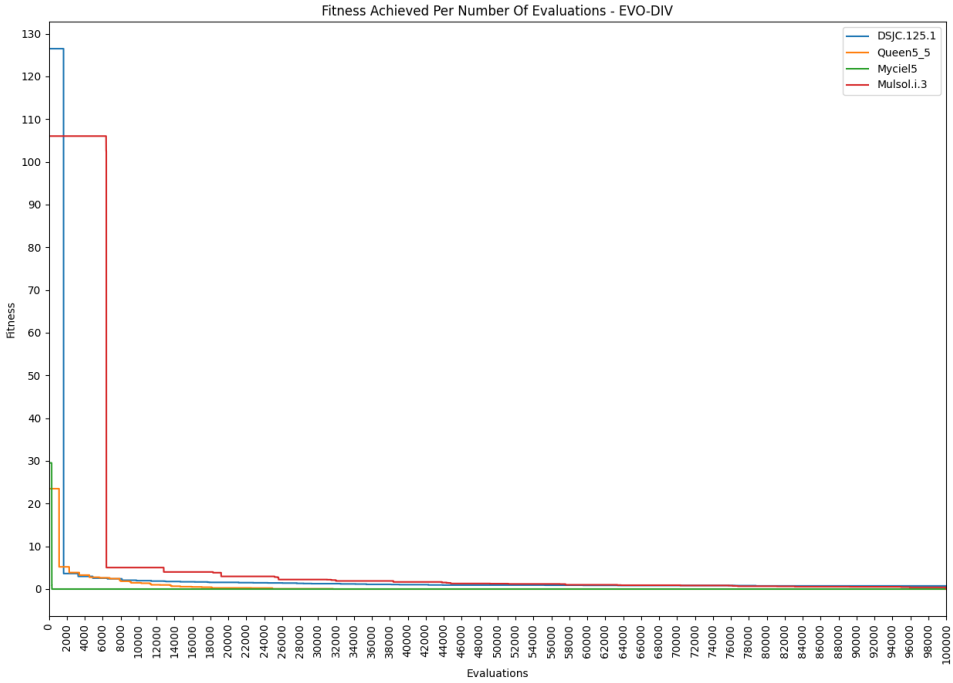
Figure 5.9: Average fitness achieved for each generation of Evocol on DIMACS test graphs

Evo-div

Evo-div was able to find colourings on all runs for the Queen5_5 and Myciel5 graphs. However on both DSJC.125.1 and Mulsol.i.3 it was stuck on the same minimum fitness value for many iterations and so was unable to find an optimal colouring in all the runs. This is probably because these two graphs were larger and once most of the population has a similar fitness value, it is a matter of random luck that the next change might reduce the fitness during the mutation or Tabu search.

Graph	Success Rates (%)	Average Final Fitness
DSJC.125.1	33.3	0.7
Queen5_5	100.0	0.0
Myciel5	100.0	0.0
Mulsol.i.3	66.7	0.4

Table 5.10: Success Rates for Evo-div on DIMACS graphs



g

Figure 5.10: Average fitness achieved for each generation of Evo-div on DIMACS test graphs

Graphs Comparison

A comparison of the performance of the different algorithms for each of the simple graphs can be seen in Figure 5.11[3].

As can be seen, the HCA and (1+1) EA are the lowest performing algorithms in general and, depending on the graph, achieve the lowest performance given the full 100000 evaluations. The exception being the queen5_5 graph, where the HCA was faster than the (1+1) EA and Evocol to consistently achieve an optimum colouring.

It is also worth noting that the (1+1) EA makes large improvements very quickly, and is actually the best performing algorithm on all of the graphs when given 1000 evaluations, in some cases maintaining a higher performance for longer than this.

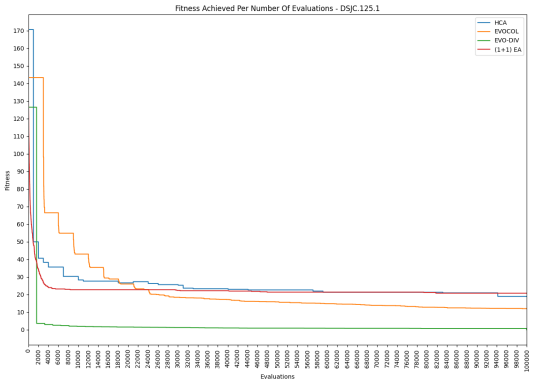
The best performing algorithm was Evo-Div, which achieved the best performance on all of the graphs we tested on.

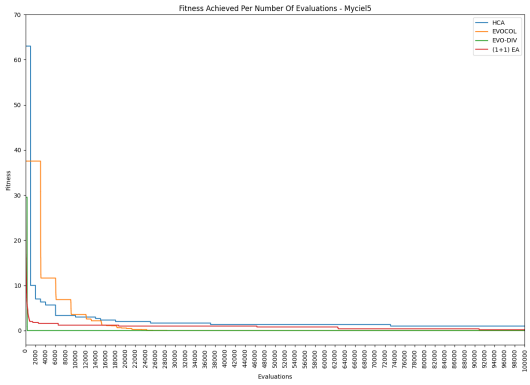
The success rates of the algorithms on the simple graphs after 100000 evaluations are shown in the table below.

Graph	(1+1) EA	HCA	Evocol	Evo-div
DSJC.125.1	0.0	0.0	0.0	33.3
Queen5_5	0.0	100.0	96.7	100.0
Myciel	0.0	36.7	100.0	100.0
Mulsol.i.3	0.0	0	0.0	66.7

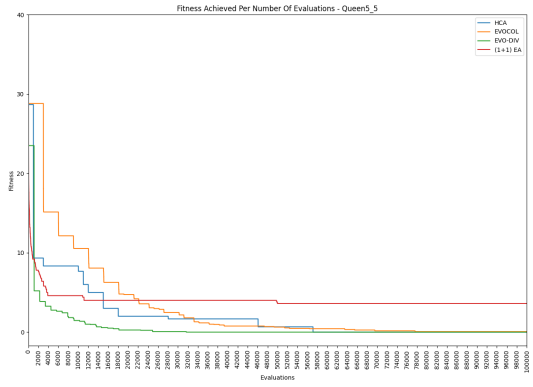
Table 5.11: Success Rates on all algorithms on DIMACS graphs

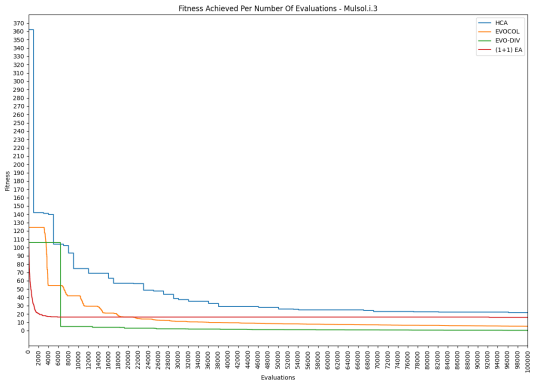
[3]Note the different scales for the y-axis of each graph in the figure





(a) DSJC.125.1 *(left)* and Myciel5 *(right)*





(b) Queen5_5 *(left)* and Mulsol.i.3 *(right)*

Figure 5.11: Comparison of the performance for each of the algorithms on the DIMACS graphs

Chapter 6

Conclusion & Future Work

6.1 Conclusion

For this project we aimed to answer the question: **how do state-of-the-art evolutionary algorithms compare to theory based algorithms when solving the graph colouring problem?** Currently, the performance of simple theory-based algorithms is well understood for simple graphs as they lend themselves well to analysis. However, on larger and more complex graphs use of these algorithms is not well documented. Additionally, complex algorithms have been tested on complicated graphs, but they are very poorly understood. As discussed, this project aimed to bridge that gap by performing a direct comparison between simple theory-based algorithms and more complex state-of-the-art algorithms, both on simple graph classes and complex, real-world graphs.

To make comparisons between the performance of the algorithms we looked at the number of **fitness evaluations** each algorithm makes to find a viable colouring for the graphs. Looking at the number of fitness evaluations is a commonly used metric in the field, as it is machine independent and is unbiased towards different complexity algorithms. Other metrics include looking at running time or looking at the number of iterations needed for an algorithm to reach a viable solution. Looking at the running time of an algorithm is very machine dependent, as hardware differences can have large effects on the values. The disadvantage with looking at the number of iterations is that is it favours algorithms with large complex iterations with many steps over simpler algorithms that only make small mutations between generations.

As well as looking at the number of fitness evaluations required to reach a valid k-colouring, we also looked at the success rate for each graph, shown in equation 5.2, where a successful run is one which the algorithm finds a proper k-colouring for the graph.

Overall we found that the complex algorithms outperformed the (1 + 1) EA and achieved lower final fitness values. The (1 + 1) EA generally reduced the number of conflicts quicker at the start of each run but eventually got stuck on most graphs and was unable to find an optimal colouring. The HCA algorithm performed best on simple graphs due its initialisation function and was able to consistently find the optimal colouring on all runs. On the DIMACS graphs the Evo-div algorithm vastly outperformed the other graphs and was able to find solutions on some runs for the DSJC.125.1 and Mulsol.i.3 graphs which none of the other algorithms were able to achieve. The (1 + 1) EA was unable to find optimal colourings on any of the DIMACS graphs. This indicates that the complex crossover and mutation functions greatly increase the likelihood of discovering the optimal colouring by exploring areas outside of the local minima. (1 + 1) EA is unable to do this as it only makes a change if it improves upon the current fitness which causes it to get stuck at a certain point and make little improvement. However, the (1+1) EA was very fast to make improvements initially, and when comparing the fitness after 1000 evaluations it shows the best performance. This suggests that the lack of crossover and complex mutation enable the (1+1) EA to initially work very well, even on complex graphs. A good approach to finding solutions to the graph colouring problem might then be to combine the (1+1) EA and Evo-div, using the (1+1) EA method until it reaches a plateau and then using Evo-div to find the optimal colouring.

6.2 Future Work

6.2.1 Simple Algorithms

The (1+1) Evolutionary Algorithm performed surprisingly well on the graphs used for evaluation and so it would be interesting to investigate the other simple algorithms described in the literature review. This would allow us to see what features of simple algorithms could be an improvement in performance.

The (2+2) Genetic Algorithm with fitness sharing, described in section 2.2, could provide interesting results,

as it introduces a simple crossover as an improvement to the (1+1) EA. It also includes a fitness sharing approach to maintain diversity. Since the (1+1) EA can realistically only make small changes each iteration, introducing a crossover function could allow bigger changes to be made which could enable it to escape from sub-optimal local minima. It has also been shown for cycle graphs that adding a simple crossover operator with an algorithm like the $(2 + 2)$ GA with fitness sharing can improve the algorithms optimisation time from $O(n^3)$ to $O(n^2)$. Additionally, the optimisation time for binary trees can be improved from $2^{\Omega(n)}$ for the (1+1) EA to $O(n^3)$ for the (2+2) GA.

Another simple algorithm, the Iterated Local Search, previously discussed in section 2.3, features a more sophisticated mutation stage. Similarly to the (2+2) GA, this would improve on the (1+1) EA as it would allow for larger changes to be made so as to allow the algorithm to escape from sub-optimal local minima.

6.2.2 State-of-the-Art Algorithms

The algorithms explored in this paper represent a small subset of existing methods for solving the graph colouring problem using evolutionary algorithms. In the future, it would be interesting to investigate the remaining theory based algorithms presented in the literature review, and also some more state-of-the-art approaches.

For example, another approach to investigate would be the AMACOL algorithm, presented by Galinier et al. in [16]. AMACOL uses what is called a 'central memory' to produce offspring in each generation, the offspring is then improved using a local search function. If the mutated offspring is then an improved solution, the central memory is updated.

Another state-of-the-art approach is the MLAMACOL, presented by Mirsaleh et al. in [17]. This is a type of Michigan memetic algorithm. In Michigan algorithms, rather than each individual representing an entire solution, each individual represents a part of a solution. In MLAMACOL each individual represents a single vertex in the graph. Each vertex is then evolved using local search operators and evolutionary approaches. It is shown that this approach outperforms many other approaches, including AMACOL described above.

A very different approach to the ones previously described is a tree-based approach described by Sharma et al. in [18], specialised to work on larger graphs. The algorithm first divides the graph by separating out smaller maximal independent sets, found using tree exploration methods. Then optimal colourings are found for each independent section before rejoining to form a solution for the full graph.

6.2.3 Graphs

It would be interesting to gain results on a wider variety of graphs from the DIMACS graphs set. Whilst we tested on different structures of graphs, it would be interesting to explore how the different algorithms fared on similar graphs of varying sizes. We mainly tested on smaller graphs so that it would be possible to gather more data, but considering larger graphs in our test set could provide some interesting data, for example the more modern approaches (Evocol, Evo-Div) might be more able to find solutions on graphs with more nodes than the older algorithms (HCA) due to containing more sophisticated features.

Chapter 7

Contributions

7.1 Daniel Marshall

I wrote up the Evocol section of the Literature Review chapter, and then proceeded to implement the Evocol algorithm (in particular the diversity control measures and MPX crossover which are unique to this algorithm), collecting raw results which were later used to create the figures in the report.

Other more minor contributions I made included writing the Introduction chapter for this report and creating some small utility programs for converting graphs from DIMACS into NetworkX format and converting between different result formats that were output by some of the algorithms.

7.2 Trisha Goel

My main contribution was implementing the Evo-div algorithm and gathering results for it. I also wrote about Evo-div and the simple algorithms in the Literature review.

7.3 Robert Dennison

My main contribution to this project was the code for the Hybrid Colouring Algorithm. I ensured that the crossover and mutation stages were working correctly and that it would output the results in a suitable format to be used in the figures in this report.

I also worked on gathering results for the HCA for each of the graphs, as well as gathering the results for the (1+1) Evolutionary Algorithm.

Other contributions include writing the Future Work section and selecting suitable graphs to make up our test set.

7.4 Sam Clowes

My main contributions to the project were to collate and analyse the raw data, including generating the graphs to display the data. Other contributions include researching to find an appropriate framework in order to allow us to implement the algorithms.

Bibliography

[1] R. M. Karp, *Reducibility among Combinatorial Problems*, pp. 85–103. Boston, MA: Springer US, 1972.

[2] A. Hertz and D. de Werra, "Using tabu search techniques for graph coloring," *Computing*, vol. 39, pp. 345–351, Dec 1987.

[3] S. Fischer and I. Wegener, "The one-dimensional Ising model: Mutation versus recombination," *Theoretical Computer Science*, vol. 344, no. 2, pp. 208 – 225, 2005.

[4] D. Sudholt and C. Zarges, "Analysis of an iterated local search algorithm for vertex coloring," in *International Symposium on Algorithms and Computation*, pp. 340–352, Springer, 2010.

[5] S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani, "Linear time self-stabilizing colorings," *Information Processing Letters*, vol. 87, no. 5, pp. 251–255, 2003.

[6] P. Galinier and J.-K. Hao, "Hybrid Evolutionary Algorithms for Graph Coloring," *Journal of Combinatorial Optimization*, vol. 3, pp. 379–397, Dec. 1999.

[7] D. C. Porumbel, J.-K. Hao, and P. Kuntz, "Diversity control and multi-parent recombination for evolutionary graph coloring algorithms," in *Evolutionary Computation in Combinatorial Optimization*, (Berlin, Heidelberg), pp. 121–132, Springer Berlin Heidelberg, 2009.

[8] D. C. Porumbel, J.-K. Hao, and P. Kuntz, "An evolutionary approach with diversity guarantee and well-informed grouping recombination for graph coloring," *Computers & Operations Research*, vol. 37, no. 10, pp. 1822–1832, 2010.

[9] F.-M. De Rainville, F.-A. Fortin, M. Gardner, M. Parizeau, and C. Gagné, "DEAP: A Python framework for evolutionary algorithms," *GECCO'12 - Proceedings of the 14th International Conference on Genetic and Evolutionary Computation Companion*, pp. 85–92, 07 2012.

[10] C. Gagné and M. Parizeau, "Open BEAGLE: A new versatile C++ framework for evolutionary computations," *GECCO Late Breaking Papers*, pp. 161–168, 01 2002.

[11] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conference* (G. Varoquaux, T. Vaught, and J. Millman, eds.), (Pasadena, CA USA), pp. 11 – 15, 2008.

[12] G. Li, "Solving the graph coloring problem with cooperative local search," Bachelor's Thesis, Karlsruhe Institute of Technology, 2016.

[13] D. J. Johnson and M. A. Trick, eds., *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, Workshop, October 11-13, 1993*. Boston, MA, USA: American Mathematical Society, 1996.

[14] R. Balakrishnan and S. F. Raj, "Connectivity of the mycielskian of a graph," *Discrete Mathematics*, vol. 308, pp. 2607–2610, 06 2008.

[15] D. Sudholt, "Crossover is provably essential for the Ising model on trees," in *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, GECCO '05, (New York, NY, USA), pp. 1161–1167, ACM, 2005.

[16] P. Galinier, A. Hertz, and N. Zufferey, "An adaptive memory algorithm for the k-coloring problem," *Discrete Applied Mathematics*, vol. 156, pp. 267–279, 01 2008.

[17] M. R. Mirsaleh] and M. R. Meybodi, "A michigan memetic algorithm for solving the vertex coloring problem," *Journal of Computational Science*, vol. 24, pp. 389 – 401, 2018.

[18] P. Sharma and N. Chaudhari, "A tree based novel approach for graph coloring problem using maximal independent set," *Wireless Personal Communications*, vol. 110, no. 3, pp. 1143–1155, 2020. cited By 0.