Graph Colouring with Small Monochromatic Components

Menaka Lashitha Bandara
Monash University, Clayton,
Victoria, Australia, 3168

Supervisor: Graham Farr
School of Computer Science
and Software Engineering

Monash University, Clayton,
Victoria, Australia, 3168

Friday June 4, 2004

Contents

1 Introduction

The classical graph colouring problem has a colourful history dating back about 150 years. This problem deals with assigning a colour to each vertex such that no two adjacent vertices share the same colour. The trivial solution to this problem is given when each vertex is given a unique colour. However, in classical graph colouring, we impose a constraint that we need to use as few colours as possible.

Initially, it was a recreational problem. What is now known as the *Four Colour Conjecture* was initially stated by *Francis Guthrie*. This conjecture states that any planar graph (map) can be coloured using at most four colours to satisfy the constraints of classical graph colouring. This conjecture was proved by *Appel and Haken* in 1976.

We focus on graph colouring with small monochromatic components. Here we relax the constraint that no two adjacent vertices can share the same colour, and rather focus on minimising the sizes of the *maximal connected monochromatic subgraphs* - maximal subgraphs in which all vertices have the same colour. Further, we focus our attention to graphs of degree ≤ 4 with only two colours.

In this project, we have implemented algorithms EF and HST, derived from the work of [1] and [2]. We have performed computational experiments from randomly generated 4-regular graphs ranging from a size of 20 to 10000. This has allowed us to study the behaviour of these algorithms with respect to average and maximum monochromatic component size, as well as with respect to time. Furthermore, analysis of the HST algorithm has allowed us to extend it to produce a new algorithm EHST.

2 Monochromatic Graph Colouring

We have previously stated that in our problem is different to the classical graph colouring problem. We have relaxed the condition that no two adjacent vertices may share the same colour. Further, we have mentioned the notion of a *maximal connected monochromatic subgraph*. In this section, we define the class of graphs and set of colours that interest us, and proceed to introduce our problem with mathematical rigour. We also discuss work that has been conducted in this field. As background knowledge, we assume that the reader is familiar with basic set theory and basic graph theory.

2.1 Our Graphs

The graphs of our interest are undirected, and connected. Throughout this document, whenever we say arbitrary graph, it should also be understood as undirected and connected.

Let $G = (V, E)$ be an arbitrary graph. Then, $V = V(G), E = E(G)$ respectively denote the set of vertices and edges of the graph G. Let $\nu_1, \nu_2 \in V$ be adjacent. Then we denote the undirected edge $\epsilon \in E$ these vertices by $\epsilon = \epsilon(\nu_1\nu_2)$. We assume familiarity with the notion of the degree of a vertex and graph.

Definition 2.1 (λ-assignment) *Let $G = (V, E)$ be an arbitrary graph. Let C denote a set of colours, with $|C| = \lambda$. Let $\gamma : V \to C$ be a map. Then γ defines a λ-assignment on G.*

In light of these definitions, we can say that the set of graphs of our interest are denoted by $\Phi_4 = \{G : degree(G) \leq 4\}$, with the colours $C_2 = \{black, white\}$ and the set of assignments $\Gamma_2 = \{\gamma : V \to C_2\}$.

2.2 Maximal Connected Monochromatic Subgraph

We assume familiarity with the concept of a connected subgraph with a property.

Definition 2.2 (Maximal Connected Subgraph) *Let $G = (V, E)$ be arbitrary. Let κ be a map, ρ be some property and let $S_\rho = (V_\rho, E_\rho) \leq G$ be a connected subgraph with the property $\rho = \kappa(V_\rho)$. Let V' be the set of vertices not in V_ρ connected via a single edge to a vertex in V_ρ. That is exactly: $V' = \{\nu \in V : \nu \notin V_\rho, [\exists \mu \in V_\rho : \epsilon(\nu\mu) \in E]\}$. If there exists no vertex $\eta \in V'$ such that $\kappa(\eta) = \rho$, then S_ρ is a Maximal Connected Subgraph (with property ρ).*

Definition 2.3 (Chromon/Monochromatic Component) *Let $G = (V, E)$ be an arbitrary graph, C a set of colours, $\kappa \in C$, and $\gamma : V \to C$. Let $\Lambda = (V_\Lambda, E_\Lambda) \leq G$ with $\gamma(V_\Lambda) = \kappa$ and Λ is maximal. Then we say that Λ is a chromon or a monochromatic component or a maximal connected monochromatic subgraph of colour κ. The size of this chromon is denoted by $|\Lambda|$ given by $|\Lambda| = |V_\Lambda|$.*

2.3 Chromon Size

We state the (obvious) fact (without proof) that for any graph G under a λ-assignment, and for each vertex $\nu \in V(G)$, we can find a chromon.

Definition 2.4 (Chromon Set) *Let $G = (V, E)$ be an arbitrary graph, C a set of colours, and $\gamma : V \to C$. Define:*

$$\Psi(G, C, \gamma) = \{\Lambda_i \leq G : \Lambda_i \text{ is a chromon }, [\forall \nu \in V(G), \exists \Lambda_j : \nu \in V(\Lambda_j)]\}$$

Then $\Psi = \Psi(G, C, \gamma)$ is the set of chromons that γ produces on G with a set of colours C.

We claim that Ψ is a unique partition of the vertex set V (Proof A.3, A.4 in Appendix A).

Definition 2.5 ($[\lambda, \kappa]$-colourable) *Let $G = (V, E)$ be an arbitrary graph. If there exists a λ-assignment $\gamma : V \to C$ such that for all $\Lambda \in \Psi(G, C, \gamma)$, $|\Lambda| \leq \kappa$, then we say that the graph G is $[\lambda, \kappa]$-colourable.*

Definition 2.6 (Ordering γ) *Let $G = (V, E)$ be some graph, with a colour set C and a λ-assignment $\gamma : V \to C$. Define:*

$$\phi(\gamma, \Psi(G, C, \gamma)) = \max\{|\Lambda| : \Lambda \in \Psi(G, C, \gamma)\}$$

Then we can say $\phi(\gamma_i, \Psi_i) \leq \phi(\gamma_j, \Psi_j)$, if $\phi(\gamma_i, \Psi(G, C, \gamma_i)) \leq \phi(\gamma_j, \Psi(G, C, \gamma_j))$

Let $|V(G)| = n$. We note that for any $\Lambda \in \Psi$, $1 \leq |\Lambda| \leq n$ (proof A.5 in Appendix A). Then, it follows that $1 \leq \phi(\gamma, \Psi) \leq n$.

We note (proof omitted - trivial):

1. That $\phi(\gamma, \Psi) = 1$ iff $\exists \kappa \in C, \forall \nu \in V(G)$ such that $\gamma(\nu) = \kappa$.
2. That $\phi(\gamma, \Psi) = n$ iff $\forall \nu, \eta \in V(G)$, if $\epsilon(\nu\eta) \in E(G)$ then $\gamma(\nu) \neq \gamma(\eta)$

We note that the first of these conditions occur when we colour all vertices of a graph with the same colour. This is trivial. The second of these is the classical graph colouring problem, if we intended to reduce the size of C to a minimum. We wish to get close to the second condition.

2.4 Our Problem

We have already mentioned the types of graphs and the colours that interest us. Now we state that we are interested in finding $\gamma \in \Gamma_2$ which which makes $\phi(\gamma, \Psi)$ small. In another interpretation, we want γ to make κ in a $[2, \kappa]$-colouring small. So, we state the problem with which we are concerned. We want to find algorithms which takes γ_i as input and produces a γ_j as output such that $\phi(\gamma_i, \Psi_i) \leq \phi(\gamma_j, \Psi_j)$.

2.5 Current work

We examine two pieces of work done by [1] and [2].

This document uses some of the semantics of [1]. Here, a simple set of operations are presented that achieve a linear bound on the size of chromons for a given graph. These operations can be used in an algorithm (EF) to transform an arbitrary assignment of colours to an assignment that usually yields a smaller chromon size.

The work by [2] is of a somewhat more complex nature. They show that it is possible to achieve a constant bound on the size of chromons for a given graph. From their proof, it is possible to derive a more complex algorithm (HST). This algorithm is not expected to yield a constant bound since the proof assumes a condition that would be algorithmically NP-hard. However, we can still expect this algorithm to perform quite well transforming an arbitrary colouring to one that is locally minimal.

Although there has been masses of work done on classical graph colouring and its generalisations, there has not been much work done on the small monochromatic components problem.

3 Algorithms

3.1 Terminology

We use some terminology with regard to the algorithms. We note that we only focus on the set of graphs Φ_4.

Definition 3.1 (Cochromatic Neighbour) *Let $G = (V, E) \in \Phi_4$ be some graph, with an assignment $\gamma \in \Gamma_2$. Let $\nu, \eta \in V$ with $\epsilon(\nu\eta) \in E$. Then, if $\gamma(\nu) = \gamma(\eta)$, then we say that η is a Cochromatic neighbour of ν.*

If a vertex is not a *cochromatic neighbour*, we say that it is *antichromatic neighbour*.

Definition 3.2 (4-balanced/balanced vertex) *Let $G = (V, E) \in \Phi_4$ be some graph, $\gamma \in \Gamma_2$, and $\nu \in V$ satisfying $degree(\nu) = 4$. Let the set of neighbours of ν be given by $N = \{\eta_i \in V : \exists\epsilon(\nu\eta_i) \in E\}$. If $\gamma(\eta_1) = \gamma(\eta_2) \neq \gamma(\eta_3) = \gamma(\eta_4)$, then we say that ν is 4-balanced (or simply balanced).*

Definition 3.3 (2-balanced) *Let $G = (V, E) \in \Phi_4$ be some graph, $\gamma \in \Gamma_2$, and $\nu \in V$ satisfying $degree(\nu) = 2$. Let the set of neighbours of ν be given by $N = \{\eta_i \in V : \exists\epsilon(\nu\eta_i) \in E\}$. If $\gamma(\eta_1) \neq \gamma(\eta_2)$, then we say that ν is 2-balanced.*

Definition 3.4 (Stable vertex) *Let $G = (V, E) \in \Phi_4$ with $\gamma : V \to C_2, \gamma \in \Gamma_2$. Let $\nu \in V$ be some vertex. If ν has more antichromatic neighbours than cochromatic neighbours, we say that ν is stable.*

We also say that a graph is stable if all its vertices are stable, and unstable if there exists a vertex that is unstable.

Definition 3.5 (Flipping colour) *Let $G = (V, E) \in \Phi_4$ with colour set $C_2 = \{black, white\}$, $\gamma \in \Gamma_2$. Let $\nu \in V$ be an arbitrary vertex. Then, define $flip : C_2 \to C_2$:*

$$flip(\gamma(\nu)) = \begin{cases} black & \text{if } \gamma(\nu) = white \\ white & \text{if } \gamma(\nu) = black \end{cases}$$

Definition 3.6 (Ripe vertex) *Let $G = (V, E) \in \Phi_4$, $\gamma \in \Gamma_2$ and $\nu \in V$. Define:*

$$\begin{aligned}
&M(\nu) = \{\eta \in V : \exists\epsilon(\nu\eta), \eta \text{ is 2- or 4-balanced}, \gamma(\nu) \neq \gamma(\eta)\} \\
&M_i(\nu) \subseteq M(\nu) : \forall\eta_1, \eta_2 \in M_i(\nu), \nexists\epsilon(\eta_1\eta_2) \in E \\
&K_1(\nu) = M_j(\nu) : |M_j(\nu)| \geq |M_i(\nu)| \\
&K_2(\nu) = M_1(\nu) \setminus K_1(\nu) \cup \{\eta \in V : \exists\epsilon(\nu\eta), \eta \text{ is not 2- or 4-balanced}, \gamma(\nu) \neq \gamma(\eta)\} \\
&K_3(\nu) = \{\eta \in V : \exists\epsilon(\nu\eta), \gamma(\eta) = \gamma(\nu)\}
\end{aligned}$$

If $|K_1(\nu)| + |K_3(\nu)| > |K_2(\nu)|$, then we say that ν is a ripe vertex.

In the following sections when we describe algorithms, we loosely use $X + Y$ to imply $X \cup Y$ and $X - Y$ to mean $X \setminus Y$.

3.2 The `stabilise` function

The stabilise function derived from [4] is used in the implementation of both algorithms. The idea behind this function is to restore the stability of a graph when the stability of the local region surrounding ν has been disrupted by flipping the colour of ν. The only vertices which can be affected (ie made unstable) by the flip are the cochromatic neighbours of ν (after the flip). We describe the algorithm:

```
function stabilise (Graph G, vertex v)
        X = { cochromatic neighbours of v }

        while not_empty (X):
                w = element (X)

                if unstable (w):
                        flip (w)

                        X = X + {cochromatic neighbours of w }
                end if

                X = X - {w}

                end if
        end while
end function
```

The function `flip_and_stabilise` as described in [4] uses this function. It needs to be emphasised that this function only works when each vertex of the graph is stable, with the exception of the local area surrounding flipped vertex.

3.3 The EF Algorithm

We describe the three operations as given in [1] and [4] that form the basis of the EF algorithm.

1. *Creating Stable graph*
   This first operation is concerned with creating a stable graph. We do this by iterating through the vertices as many times as necessary until there are no more unstable vertices.
2. *Balanced vertices*
   Let ν, η be balanced adjacent vertices, which are antichromatic. Then we flip ν and η.
3. *Reducing chromon sizes*
   Let ν be a balanced vertex. We flip ν if doing so makes it belong to a smaller chromon.

We state the EF algorithm as it appears in [1] and [4]. We assume that the graph $G = (V, E) \in \Phi_4$ and we have arbitrarily assigned a colour to each $\nu \in V$.

```
function ef_algorithm (Graph G)
        while operation 1, 2, or 3 can be done somewhere in G
                if operation 1 can be done somewhere in G, then
                        do it
                else if operation 2 can be done somewhere in G, then
                        do it
                else if operation 3 can be done somewhere in G, then
                        do it
        end while
end function
```

To efficiently implement EF, we present the following analysis. We note that the condition of stability is very important in this algorithm. These operations need to be applied in order. One precondition at stage (2) and (3) is that the graph needs to be stable. However, when (2) is performed, we are potentially making the graph unstable. Consider what happens to the cochromatic neighbours of the vertices that are flipped. It is important that after performing (2) that we re-stabilise the graph to the effect of (1). Similarly, operating on

the graph by (3) may cause the state of the graph at the end of (2) and (1) to be changed undesirably. Then, we need to carry out an operation to the effect of (1) and (2).

The operations of re-stabilising the graph after (2) and (3) can be done locally for efficiency. Since we are guaranteed of the stability each vertex with the exception of local region around a flip, we can achieve this by simply looking at the cochromatic neighbours of the vertices we flip. We look at these same vertices to perform local operations to the effect of (2).

The implementation of the EF algorithm is discussed in Appendix F.2.

3.4 The HST Algorithm

The HST algorithm is a derivation from [2]. The proofs presented in [2] are not explicitly algorithmic. The derived algorithm is given explicitly in [4]. Throughout this section $G = (V, E) \in \Phi_4$.

Prior to our discussion of the HST algorithm, we describe a function `flip_and_stabilise()` as defined in [4]. The preconditions are: the vertex it flips is 2 or 4-balanced, and the colouring of the graph is stable. The post condition is a graph that is stable with the colour of the intended vertex flipped.

```
function flip_and_stabilise (Graph G, vertex v):
        flip (v)

        stabilise (G, v)
end function
```

We note that as with the EF algorithm, the first step in this algorithm is to create a stable colouring of G. We then attempt to minimise the chromon size of white vertices without giving regards to the component size of the black vertices. This is achieved by operating on the ripe vertices of the graph, then the white balanced vertices. We deliver this part of the algorithm as a function `initialise_colouring()`. Note that in this function, $K(\nu) = K_1$ (as defined in Definition 3.6).

```
function initialise_colouring (Graph G):
        while unstable (G)
                do operation 1 from EF algorithm
        end while

        while there exists ripe vertex or
                        white balanced vertex in G:

                if there exists v, a ripe vertex:
                        w = a vertex in K(v)

                        flip_and_stabilise (w)
                else if there exists v balanced, white:
                        flip_and_stabilise (v)
                end if
        end while
```

Given any graph G arbitrarily coloured, this function outputs a graph that is stable and contains white chromons of size not more than 2. Since the colouring is stable, we can be guaranteed that the black chromons are always either paths or circuits.

Since `initialise_colouring()` does not attempt to minimise the size of the black chromons, we may have effectively increased κ in our $[2, \kappa]$-colouring. In order to reduce κ, we need to flip some black vertices. The following function `black_search()` searches the black chromons for a set of candidate vertices to flip.

```
function black_search (Graph G):
        S = {}

        for each B = black chromon:
                if B is a circuit:
                        S = S + { every second vertex of B, but no two vertices adjacent }
                else if B is a path:
                        S = S + { every second vertex of B, but excluding end vertices }
        end for

        return S
end function
```

If we consider the set of vertices S as not being coloured either white or black, then effectively, all black vertices also have chromon size not more than 2. So, our colouring of $G \setminus S$ is now a $[2, 2]$-colouring. (or we can consider the them to coloured "grey" - but this would make it a $[3, 2]$-colouring of G). No matter which colour we assign to each vertex in S, it will make $\kappa \geq 2$ in our final $[2, \kappa]$-colouring. So we need to make this choice in a way that would make κ small.

We achieve this by creating two directed subgraphs from the vertex set S. We create an edge between two vertices depending on their distance with respect to black or white vertices. That is, we say that vertex ν, η are distance d apart over black vertices if there exists a path from ν to η such that we encounter no white vertices. Similarly we can define distance over white vertices. The structure of these graphs are important in deciding whether we colour each vertex in S black or white. We denote these graphs in the following function by $H[1]$, and $H[2]$.

We also claim that operating on the graph by `initialise_colouring()` makes the graphs $H[1]$ and $H[2]$ contain only disjoint paths or circuits. This is a crucial property that we exploit in giving these paths and circuits a direction. A directed path or circuit (of size > 1) possesses an important property. Either a vertex has a unique successor or predecessor (or both). There exists no successor or predecessor if our subgraph has size 1.

The following function `assign_final_colouring()` is an algorithm that partitions S into vertices that should be coloured black and white.

```
function assign_final_colouring (Graph G, vertex S):
        Graph H[1] = (S, E1),
                E1 = { edge (uv): u, v in S,
                        and distance (u, v) = 2 or 3 over white vertices }

        Graph H[2] = (S, E2),
                E2 = { edge (uv): u, v in S,
                        and distance (u, v) = 2 or 3 over black vertices }

        for each vertex v in H[1]:
                find beginning of path or circuit that v belongs to
                give that path or circuit a direction
        end for

        for each vertex w in H[2]:
                find beginning of path of circuit that w belongs to
                give that path or circuit a direction
        end for



        X[1], X[2] = {}
        U = S

        u = some vertex in U
        U = U - {u}
```

```
        X [1] = X [1] + {u}
        i = 1

        while not_empty (U):
                if u has a successor w in H [i] unplaced:
                        X [3 - i] = X [3 - i] + {w}
                        U = U - {w}
                        u = w
                        i = 3 - i

                else if  u has a predecessor in H [i] unplaced:
                        X [3 - i]  = X [3 - i] + {w}
                        U = U - {w}
                        u = w
                        i = 3 - i

                else
                        u = some vertex in U
                        U = U - {u}
                        X [1] = X [1] + {u}
                        i = 1
                end if
        end while

        for each vertex v in X[1]:
                colour (v) = white
        end for

        for each vertex v in X[2]:
                colour (v) = black
        end for
end function
```

We see that the two sets $X[1]$ and $X[2]$ contain the vertices that are to be coloured white and black respectively. We choose the vertices that belong in each of these sets by alternating between the two subgraphs and checking whether we have placed successor/predecessor vertices of each vertex in U already in either set. The alternating between the graphs permits us to achieve a small κ in our final $[2, \kappa]$-colouring of G.

We state the HST algorithm combining these three functions together:

```
function hst_algorithm (Graph G):
        initialise_colouring (G)
        S = black_search (G)

        assign_final_colouring (G, S)
end function
```

Implementation specifics are given in Appendix F.3.

3.5 The EHST (LASHI) Algorithm

The EHST algorithm is essentially an extension of the HST algorithm. There are three main features that make it distinct from HST. Since the bulk of the algorithm is identical to the HST, we will refer to section §3.4 and only present the changes.

The first difference is the `initialise_colouring()` function. We call the EHST version `einitialise_colouring()` and describe it:

```
function initialise_colouring (G, v):
        while unstable (G)
                do operation 1 from EF algorithm
        end while

        while there exists ripe vertex or
                        white balanced vertex in G:

                for each vertex v in G:
                        if v is a ripe vertex:
                                w = a vertex in K(v)

                                flip_and_stabilise (w)
                        else if v balanced, white:
                                flip_and_stabilise (v)
                        end if
                end for
        end while
```

This only contains a subtle difference from the function in §3.4. In §3.4, we only look at white balanced vertices iff there exists no ripe vertex. When ever we flip a white balanced vertex, it may create a ripe vertex, so we need to look at all the ripe vertices again.

Our implementation paves the way for greater optimisation during implementation. We look at each vertex, and if it is ripe, we perform the ripe operations, and if not, we check whether it is white and balanced. So in effect, our implementation permits a white balanced vertex to be flipped while there could still be ripe vertices in the graph. We conjecture that `einitial_colouring()` is functionally equivalent to `initial_colouring()`. We leave this unproven.

The second function to which we make a subtle amendment to is `assign_final_colouring()`. This amendment is made in the `while` statement. We call our version `eassign_final_colouring()` and define it below (note we only give the `while` statement - all other parts are identical).

```
function eassign_final_colouring (Graph G, vertex S):
        .
        .
        .
        while not_empty (U):
                if u has a successor w in H [i] unplaced:
                        X [3 - i] = X [3 - i] + {w}
                        U = U - {w}

                        if u has a predecessor z in H [i] unplaced:
                                X [3 - i] = X [3 - i] + {z}
                                U = U - {z}
                                u = z
                        else
                                u = w

                        i = 3 - i

                else if  u has a predecessor in H [i] unplaced:
                        X [3 - i]  = X [3 - i] + {w}
                        U = U - {w}
                        u = w
                        i = 3 - i

                else
                        u = some vertex in U
                        U = U - {u}
                        X [1] = X [1] + {u}
```

```
                        i = 1
                end if
        end while
        .
        .
        .
end function
```

Unlike in §3.4, we ensure that when a vertex ν has both a predecessor and successor in $H[i]$, we give the predecessor and successor the same colour opposite that of ν. From analysis of local regions for when the HST produced component sizes > 6, we find that this alteration eliminates this undesirable effect most of the time.

The last change is to add a new function, `minimise_chromons()`. We define this function:

```
function minimise_chromons (Graph G)
        for each vertex v in G:
                if flipping v would make it belong to a smaller chromon:
                        flip (v)
                end if
        end for
end function
```

Analysis of the HST implementation revealed that the HST algorithm could do better if it were to perform an operation similar to the third operation of the EF algorithm. Further analysis of the local regions around chromons of size > 6 produced by the HST algorithm showed that it could do better by flipping some of the vertices. It is important to note, however, that after the graph has been operated on by `minimise_chromons()`, we can no longer guarantee that the components will be paths and circuits. That is, we have not looked closely enough at the graph after this step to determine whether the colouring is stable.

We now give a top-level description of the EHST algorithm:

```
function ehst_algorithm (Graph G):
        einitialise_colouring (G)
        S = black_search (G)

        eassign_final_colouring (G, S)

        minimise_chromons (G)
end function
```

Some implementation details are given in Appendix F.4.

4 Random Graph Generation

We discuss the second algorithm presented in [3] and [5] for generating 4-regular graphs that are approximately uniformly distributed. The following discussion is an elaboration on [3] and [5].

The graphs we wish to generate are are undirected. Thus, we know that an undirected, arbitrary graph of size n of degree n can have at most $\frac{n(n-1)}{2}$ possible edges. Although we only deal with 4-regular graphs, in order for us to place edges uniformly, we need to look at all these possible edges. We call a possible edge a pseudo-edge, which is simply a pair of vertices denoted by $\mu\eta$ where μ, η are vertices. For each pseudo-edge we associate a weight:

$$r_{\mu\eta} = (4 - degree(\mu))(4 - degree(\eta))$$

Let S be the set of all allowed pseudo-edges. We define a normalisation variable R:

$$R = \sum_{\mu\eta \in S} r_{\mu\eta}$$

Note that: $R = 0 \Longleftrightarrow [r_{\mu\eta} = 0, \forall \mu, \eta : \mu \neq \eta]$.

We define the probability of making a pseudo-edge an edge:

$$Pr\left(\mu\eta\right) = \frac{r_{\{\mu,\eta\}}}{R}$$

When choosing an edge, the following constraints need to be satisfied:

1. If an edge $\epsilon(\mu\eta)$ exists in the graph, then we disallow placing an edge $\eta\mu$.
2. We can never have an edge $\mu\mu$.

We create a list L from the items in S so that $L = [\mu_0\mu_1, \mu_0\mu_2, \ldots, \mu_{n-1}\mu_n]$. We denote element i of L by: $element(i, L)$. Then, for element i in the list, we define I_i inductively:

$$I(0) = Pr(element(0, L))$$
$$I(n+1) = I(n) + Pr(element(n+1, L))$$

Since $\sum Pr(\mu\eta) = 1$, then $\sum(I_{i+1} - I_i) = 1$. Initially, all pseudo-edges have equal probability.

We state (our) version of the algorithm:

```
function random_graph (size n)
        R = 16 * n * (n - 1) / 2
        L = [ List of pseudo-edges ]

        while ( R > 0 ):
                U = uniform_random_number in [0,1)

                pick the pseudo-edge element(e, L) such
                        that I(e) <= U < I(e+1).

                place edge e in G
                remove(e, L)

                Recompute R.
                Recompute I(e) for element(e,L).
        end while

        Output: G a graph.
end function
```

By the property we stated earlier, when $R = 0$, we must have all $r_{\mu\eta} = 0$. In some instances, however, this does not guarantee that the graph is 4-regular. Thus, it is important that we check whether we have $2n$ edges in our graph. If not, then we need to try again.

Some implementation details are provided in Appendix E.

5 Guide to using the software

5.1 Using `graphwrite`

The `graphwrite` utility allows the user to create uniformly distributed, random 4-regular graphs which can consequently be used as input for the `process` utility. It allows for graph creation to be parallelised to allow clusters to create large graphs which would be time consuming as a serial operation.

Its interface is primitive: `graphwrite n k output_filename`

The first two arguments are mandatory. The command above would create k graphs of size n. If the last argument (i.e., output file) is omitted, then the graphs will be written to the console.

As an example, the following command: `graphwrite 3000 10 3000-10.graph` would create 10 graphs of size 3000 to file `3000-10.graph`. A more complex example would be where a cluster of nodes with hostnames `node0` to `node3` will be used to produce a total of 12 graphs for size 10000:

```
ssh node0 graphwrite 10000 4 10000.0
ssh node1 graphwrite 10000 4 10000.1
ssh node2 graphwrite 10000 4 10000.2
ssh node3 graphwrite 10000 4 10000.3
```

All errors detected by the program will output some useful message. These errors can generated at many nested levels, and are handled by the exception handling environment. These errors will not be listed here. If core-dumps are enabled, a core file will usually be created. However, it is worthwhile noting a common error that the user may encounter:

```
generator.c: 91: initialise failed to allocate memory to node edge_list
Aborted
```

This error means that there is not enough physical memory to create the graph of the specified size. This is not a bug in the software. The only remedy is to increase the physical memory.

Format of graph files are discussed in Appendix C.

5.2 Using `process`

The process utility runs the three algorithms on a set of graphs and reports various statistics for each graph and a summary for each algorithm.

It has a minimal command line based interface: `process options`.

The `options` are given by:

- `--help` or `-h`
  Prints out a simple help screen to to allow a user to quickly understand the basics of the utility.
- `--version` or `-v`
  Prints out version information, compiler information, and optimisation information.
- `--input graph` or `-i graph`
  Read graphs from a file named `graph`.

- `--output stats` or `-o stats`
  Output information to filename `stats`.
- `--max-limit k` or `-m k`
  Issued with `-i`, this reads at most k graphs from the file. It reads $< k$ graphs if it encounters end-of-file first. Without $-i$, the software randomly generates k graphs.
- `--size n` or `-s n`
  Generate graphs of n vertices.

If no options or only `-m` is specified, then `process` will interactively ask the user to input the size of graph to produce. If `-m` is omitted, then the behaviour is equivalent to if `-m 1` was issued. If `-i` is specified without `-m`, the all the graphs in the specified file will be used.

If both `-i` and `-s` are specified, then the former argument will be given precedence. That is, `process` would always prefer to read graphs from a file than to create them randomly.

The following example illustrates reading in graphs from a file `k-graph` and outputting results to `k-graph.results`:
`process -i k-graph -o k-graph.results`

We randomly create 25 graphs of size 1000 and the results are written to the console:
`process -s 1000 -m 25`

We read the first 3 graphs from file `graphs` and output to `graphs-3.results`:
`process -i graphs -m 3 -o graphs-3.results`

This software, on error, will produce useful messages. Thus, the list of possible abnormal exit conditions will not be discussed here. Among the more common are parse errors in a malformed graph file. Furthermore, if a single graph (it may be the last graph) is defined incorrectly, then `process` will abandon its efforts and will refuse to report results.

6 Testing

In order to build a stable and correct system, testing was performed according to the scheme described below. It is important to note that the following scheme works as a consequence of the software development model. The code is also written such that during compilation, certain flags could be enabled which includes more code for debugging and verbose output. This is explained in Appendix B. We describe our testing scheme:

1. All primitive functions and methods were given some strictly defined behaviour on all inputs. That is, boundary conditions were correctly identified, and invalid input was correctly dealt with, along with the general behaviour.
2. These primitive functions and methods were then tested on these inputs. It was ensured that the correct behaviour was produced on these inputs.
3. Higher order functions and methods are built from the primitive ones. These functions have a logical behaviour defined in terms of the behaviour of the constituent primitive functions which are known to be correct. Thus, we reason that the emergent behaviour must also be correct. Test cases were also thrown at these higher order functions for the purpose of verification.

Assertions are only enabled when debugging flags are enabled during the compilation of software. They are disabled by default for efficiency.

In the source tree, the `testing` directory includes various code used for testing. It is important to note that some of the methods defined in the data types are given for completeness. They have not been used in this software. Due to the lack of time, these methods have not been tested thoroughly, and no guarantee can be made that they will behave correctly. However, all methods of the data types used in the software have been tested throughly.

In algorithm testing, the constituent parts were initially tested with known inputs. Then, the whole algorithm was tested on some small graphs to verify that they produced expected behaviour.

7 Computational Experiments

7.1 Creating Graphs

Graph creation takes the most amount of space and time. For instance, in order to produce a graph of 10000 vertices, the generator needs $763MB$ memory from the heap. So the approach taken was to create graphs using the `graphwrite` utility and store them into a file.

We created graphs of size $20 - 100$ in steps of 20, $100 - 1000$ in steps of 100 and $1000 - 10000$ in steps of 1000. For each of these sizes, we created 25 graphs. The Monash Parallel Parametric Modelling Engine (PPME) cluster was the main piece of hardware used to create the graphs. The following is a more complete description of the apparatus. Note that some node machines on the cluster were SMP and others non-SMP. The output describing hardware is extracted from the `dmesg` of the Linux kernel for accuracy.

- Hostname of (used) nodes: node01-node15, node22, node23, node25, node33.
- CPU0: Intel(R) Pentium(R) 4 CPU 3.00GHz stepping 09 (Non-SMP)
  CPU1: Intel(R) Pentium(R) 4 CPU 3.00GHz stepping 09 (SMP)
- Memory: 1031356k
- Linux kernel version 2.4.22-xfs (non-SMP) or 2.4.26-1-686-smp (SMP)
- Operating System: Debian GNU/Linux testing/unstable.
- Libc6 version: 2.3.2.ds1-12
- Compiler: gcc (GCC) 3.3.3 (Debian 20040401)

The following was the method employed in graph production:

1. For small graphs ($20 - 1000$ vertices), the job was launched onto a single node with the command:
   `ssh nodei graphwrite k 25 k-run`
   where k was the size of the graph, and i the number of the node.
2. For larger graphs ($2000 - 7000$ vertices), this workload was spread over the nodes - that is, each graph size was assigned to a unique node, with all 25 graphs produced by that node.
3. For the very large graphs ($8000 - 10000$ vertices), the workload was spread out over about 15 nodes, with each node producing $2 - 3$ graphs. These graphs were combined into one file with the command:
   `cat k-run.1 k-run.2 k-run.3 ... k-run.n > k-run`
4. Random number generation was done via the function `drand48()`. Random seeding was done by reading from the random source `/dev/random`.

7.2 Running Algorithms

The algorithms take a considerably small amount of time than the graph generation procedure. Hence, slower hardware was used to measure the performance - in an attempt to increase the resolution of the measure.

The following were the apparatus used to test the algorithms:

- Processor: Motorola 745/755 600MHz PowerPC G3 (Revision 51.17 (pvr 0008 3311))
- Memory: 384MB
- Linux kernel 2.6.6

- Operating System: Debian GNU/Linux Unstable.
- Libc6 version: 2.3.2.ds1-12
- Compiler: gcc (GCC) 3.3.3 (Debian 20040422)

The following procedure was used to test the algorithms:

1. All time measurements were done by calling `getrusage()` system call, which allows accounts for kernel level preemption.
2. For each file `graphfile` containing the graphs of the same size, the following command was run: `process -i graphfile -o graphfile.results`.
3. The script `createplot.sh` was run on each output file produced by `process` to produce a Gnuplot plot file. Then for each algorithm EF, HST, EHST respectively, the following command was issued in the directory where the result files were stored:
   `createplot.sh EF > plot-ef.data`,
   `createplot.sh HST > plot-hst.data`,
   `createplot.sh EHST > plot-ehst.data`.
4. Then
   `texproduce plot-ef.data > plot-ef.tex`,
   `texproduce plot-hst.data > plot-hst.tex`,
   and `texproduce plot-ehst.data > plot-ehst.tex`
   was issued to produce a LaTeX table.

7.3 Results

The tables of results are given in the Appendix G.

7.4 Error

Our main concern with errors in our measurement is with measuring the time taken by the algorithms.

We have used RISC hardware since we believe that the calculated time better reflects the number of elementary instructions in the algorithm. Furthermore, we have attempted to minimise error resulting from software by using the `getrusage()` function to compute the time each algorithm has spent on the processor. This is a more accurate measure than time-stamping (ie, using `gettimeofday()`) since this deals with acquiring information from the kernel about how much time a process has spent on the processor. Furthermore, we only take into account time spent in user mode since it was assumed that time spent system mode is irrelevant to the actual functionality of the algorithm. It can be seen from the standard deviations given in the tables in Appendix G that `getrusage()` is an accurate measure. For small graphs, however, we find significant error. This is most likely due to the fact that the hardware is too quick to be able to measure the algorithms effectively for small graphs.

7.5 Discussion

We used a sample space of 25 graphs for each size of graph. We varied the size of graphs from between 20 to 10000. Our analysis will primarily focus on the large scale behaviour of the algorithms. We discuss in detail four particular aspects measured, which can be found in the tables given in Appendix G.

We know from [1] that the EF algorithm is bounded sub-linearly in terms of maximum monochromatic component size. The curve in Figure 1 illustrates this. We hypothesise that the HST implementation may also

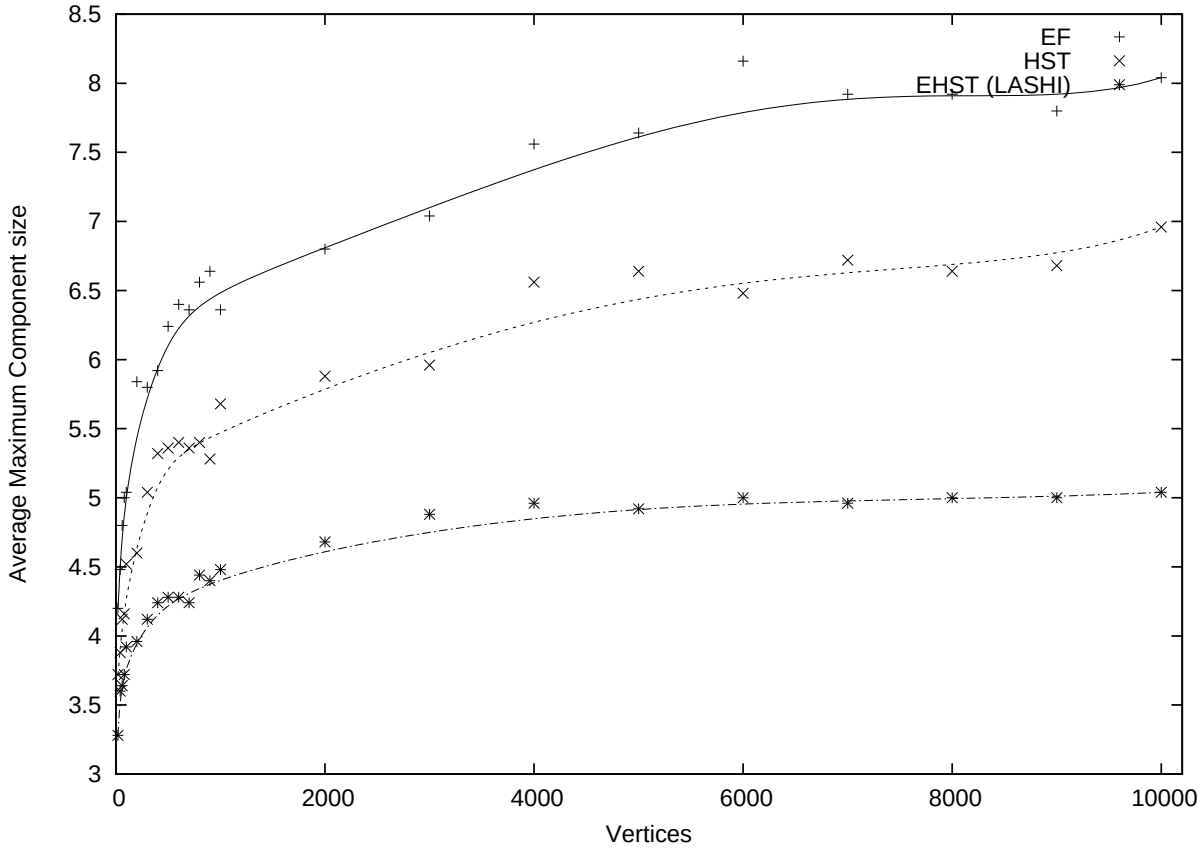

Figure 1: Average Maximum Component size

bounded sub-linearly since it exhibits similar behaviour to the curve produced by the EF algorithm. This hypothesis is consistent with figure 2, where we see a maximum monochromatic component size of 7 from a size of 300 to 2000, and a sudden jump to 8 for > 2000. Following this pattern, we might expect that at some graph size $k > 10000$, the maximum monochromatic component size will jump to > 8.

We also note that the EF and HST algorithms exhibit slightly erratic behaviour in terms of maximum chromon size. The standard deviation values (in Appendix G) for these two algorithms seems to suggest this. Figure 2 shows particularly well how the maximum monochromatic component size jumps erratically for the EF algorithm.

Unlike EF and HST, Figure 1 illustrates that the maximum monochromatic component size for the EHST algorithm is likely to be bounded by a constant. The curve here looks "more" logarithmic than for the other algorithms. This is also reflected for the values in the table, which shows that the average maximum monochromatic component size is reaching some limit < 6. Unlike the EF and HST algorithms, the data points here are well behaved, and their variation is small. We see that the standard deviation (in Appendix G) for the maximum monochromatic component size is smaller than that of the other two algorithms. Thus, we claim that the EHST algorithm increases in maximum chromatic component size logarithmically. Furthermore, from our analysis, we conjecture that the maximum monochromatic component size is bounded above by 6.

Figure 3 illustrates that all algorithms converge to some minimum with respect to average monochromatic component size. It is interesting to note that the average monochromatic component size of the HST algorithm is lower than the EHST algorithm, although the EHST algorithm performed much better in terms of maximum monochromatic component size. There is only one conclusion that could be drawn from this - the EHST algorithm makes a trade off between the average monochromatic component and the maximum monochromatic component size to reduce the latter. In other words, the EHST algorithm favours a lower maximum monochromatic component size over a lower average monochromatic component size. The EF algorithm performs quite poorly here, and the difference between HST and EF is much larger than HST and EHST as seen in figure 3. This, however, *disproves* an erroneous conclusion that could have been made

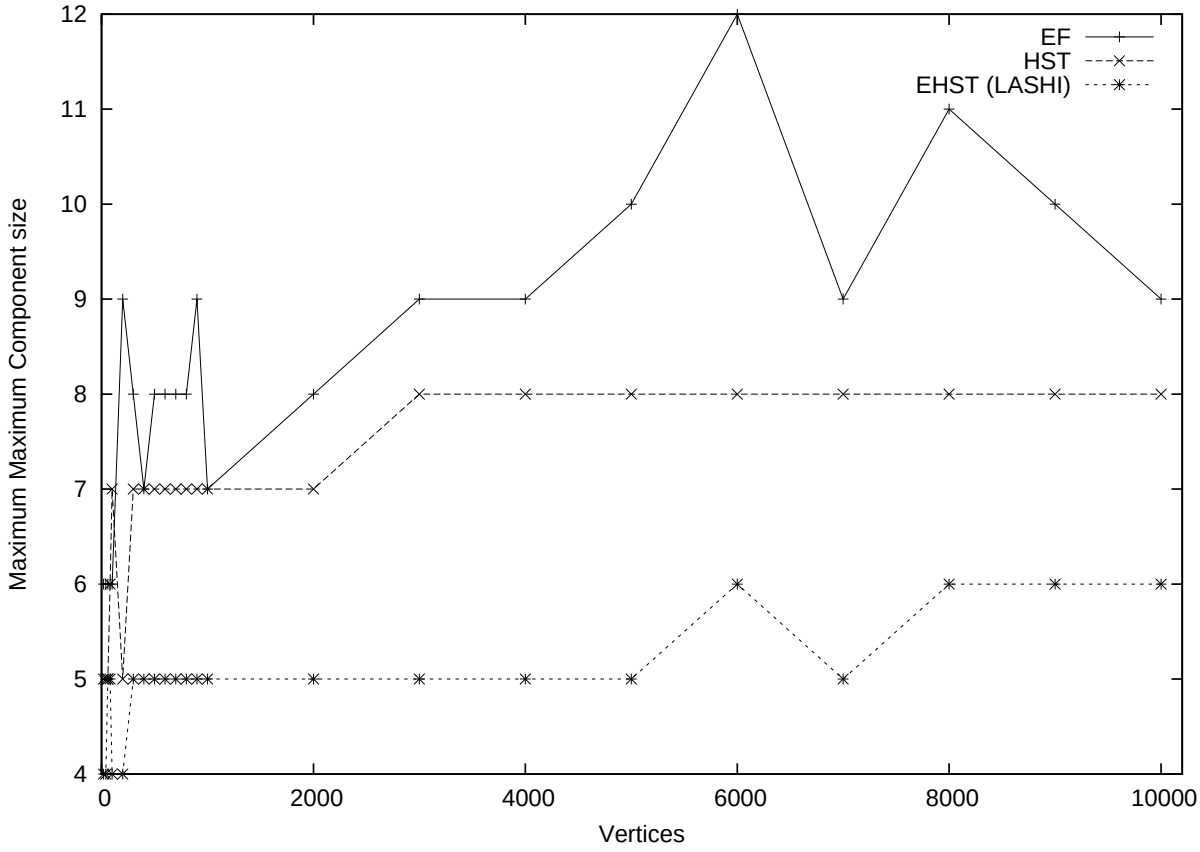

Figure 2: Maximum Maximum Component size

from our discussion of the HST as compared with EHST. That erroneous conclusion is: A smaller average monochromatic component size implies a larger maximum monochromatic component size. This is not true in general, as the EF algorithm illustrates. In fact, we can say that EHST *chooses* to favour a smaller maximum component size over a smaller average component size. There may exist an algorithm whereby both a lower maximum and average component size may be achieved.

The running time of the algorithms are illustrated in Figure 4. At a glance, these graphs show that all three algorithms are likely to be of complexity $O(n^2)$. We can say with certainty that they are non-linear. The quickest of these is the EF algorithm. We see that the HST and EHST algorithms are slightly slower than the EF algorithm, and that the EHST algorithm is slightly slower than the HST. This is expected since the EHST algorithm has an extension to the HST and does some extra work which is linear time bounded.

Although we previously discussed the possible error introduced in computing these timings, we have quite specifically focused on the large scale behaviour of the algorithms. From our results tables, we find that for large n, the standard deviations are small compared with the magnitudes of the times. Furthermore, as n increases, this ratio becomes smaller, which suggests that our measure becomes *more* accurate as we look at larger and larger graphs. Hence, we can conclude that our measurements are accurate and depict the true behaviour of these algorithms with respect to time.

7.6 Conclusion

We can see that the EF algorithm works quite well for graphs of up to 1000 vertices in comparison with the HST algorithm. This is a surprising result since the EF algorithm works with a much simpler set of operations as compared with the HST algorithm. Even for graphs > 1000 vertices, we find that the EF algorithm performs reasonably well, although its behaviour in terms of maximum monochromatic component size is slightly more erratic than that of the HST algorithm.

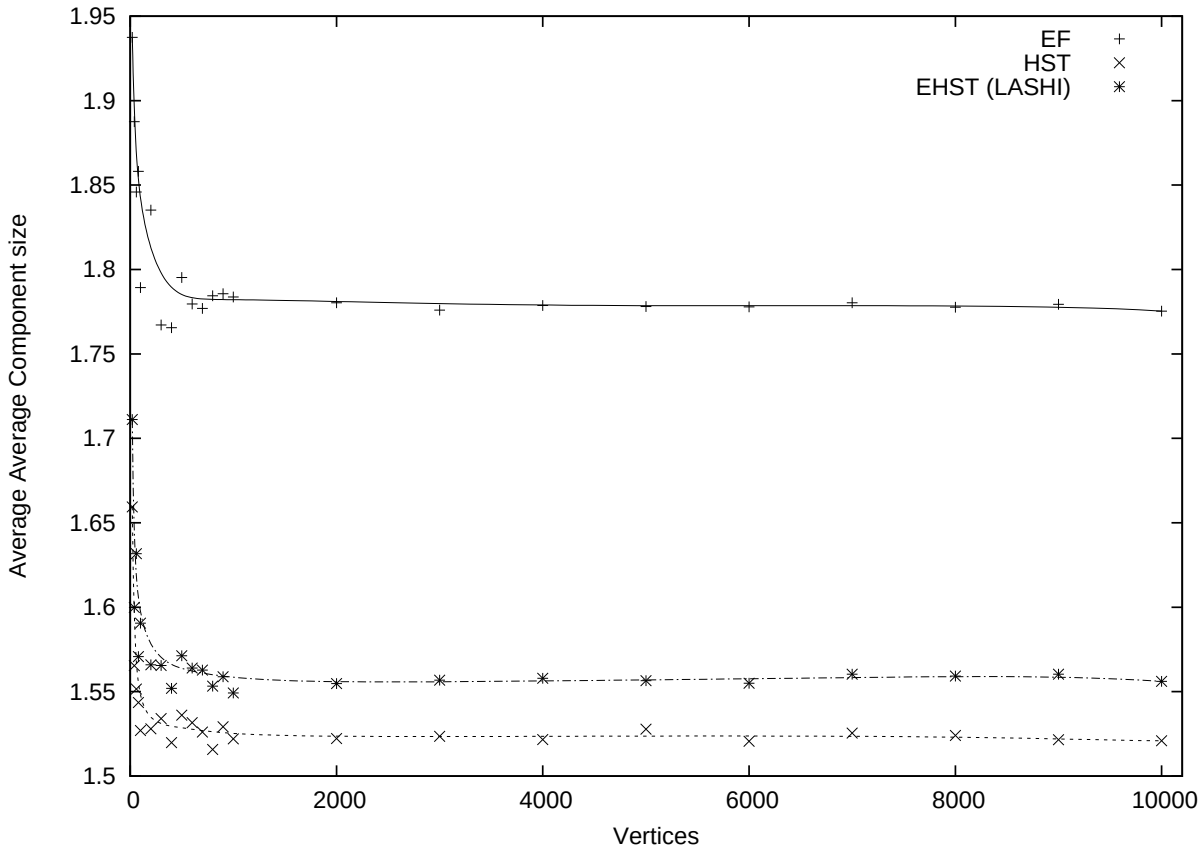

Figure 3: Average Average Component size

The EHST algorithm outperforms these, and is quite well behaved for all the graphs tested. This algorithm always yielded a maximum component size of ≤ 6. So, we conjecture that the EHST algorithm is in fact bounded above by a constant of 6 in terms of maximum component size. This is probably the most important result to come through from this experiment. We know from [2] that such algorithms can exist. However, a direct conversion of the proof in [2] to an algorithm requires solving a NP-Hard problem to achieve this constant bound. The simple extensions to the HST which yielded EHST has achieved this for all the graphs tested.

There is some interest which has been a consequence of this work. Primarily, the EHST algorithm should be pursued further. It would be interesting to mathematically prove that this does in fact achieve what [2] claims since this would give a polynomial time algorithm to one that requires solving an NP-Hard problem. However, it is anticipated that this may actually be difficult since EHST no longer guarantees that components are paths or circuits - a property which has elegant mathematical features. Furthermore, it would be worthwhile to perform computational experiments on larger graphs. It would also be interesting to see whether similar extensions could be done to the EF algorithm which would yield better results. It may also be possible extend HST algorithm further to work as effectively as EHST in terms of maximum component size while keeping the average component size at least as low as HST.

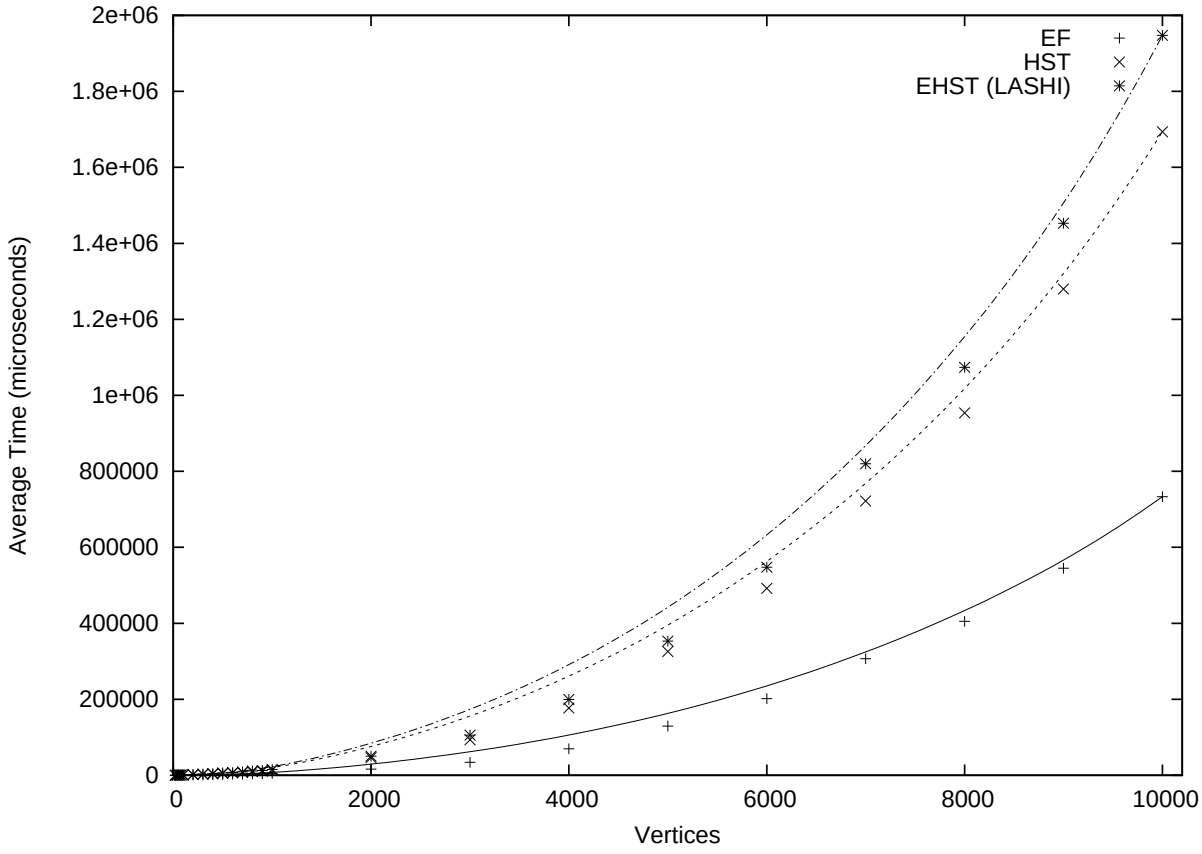

Figure 4: Average Time

A Appendix - Proofs of some results

Lemma A.1 *Let G be some arbitary graph, C a set of colours, and $gamma : V(G) \to C$. Let Ψ denote the set defined in Definition 2.4. Then $\Psi \neq \emptyset$.*

Proof

We assume that $V(G) \neq \emptyset$. The, we can find $\nu \in V(G)$. Let $\gamma(\nu) = \kappa \in C$. Then we are guarenteed to find a chromon $\Lambda \in \Psi$ such that $\nu \in V(\Lambda)$. Then $\Lambda \neq \emptyset$ which implies $\psi \neq \emptyset$.

□

.

Lemma A.2 *Let G be an arbitary graph, C a set of colours, and $\gamma : V(G) \to C$. Let Ψ denote the set defined in Definition 2.4. Let $\Lambda_i, \Lambda_j \in \Psi$. Then*

$$\nu \in V(\Lambda_i), \text{ and } \nu \in V(\Lambda_j) \iff \Lambda_i = \Lambda_j$$

Proof ($\Longleftarrow$)

Trivial. If $\Lambda_i = \Lambda_j$, then it neccessarily follows that $V(\Lambda_i) = V(\Lambda_j)$. Then, $\nu \in V(\Lambda_i) \iff \nu \in V(\Lambda_j)$.

Proof ($\Longrightarrow$)

We note that if $V(\Lambda_i) = V(\Lambda_j) \implies E(\Lambda_i) = E(\Lambda_j)$. since $\Lambda_i, \Lambda_j \leq G$.

Suppose that $\nu \in V(\Lambda_i)$ and $\nu \in V(\Lambda_j)$ but $\Lambda_i \neq \Lambda_j$.

Let $\eta \in V(\Lambda_i), \eta \notin V(\Lambda_j)$, satisfying $\epsilon(\nu\eta) \in E(\Lambda_i) \subseteq E(G)$. This must exist, since if it did not, then Λ_i would be a disconnected subgraph, which we know is a contradiction.

Let $\gamma(\nu) = \kappa, \exists k \in C$. Then $\gamma(V(\Lambda_i)) = \gamma(V(\Lambda_j))$, which implies that $\gamma(\eta) = \kappa$.

Then we have $\nu \in V(\Lambda_j)$ with $\eta \notin V(\Lambda_j)$ and $\epsilon(\nu\eta) \in E(G)$, and $\gamma(\nu) = \gamma(\eta)$. This implies that Λ_j is not maximal, which is a contradiction.

So, $V(\Lambda_i) = V(\Lambda_j) \implies E(\Lambda_i) = E(\Lambda_j) \implies \Lambda_i = \Lambda_j$.

□

Proposition A.3 (Partition of a Graph) *Let G be an arbitrary graph, and C a set of colours, and $\gamma : V(G) \to C$. Let Ψ denote the set defined in Definition 2.4.*

Proof

Lemma A.1 shows that $\Psi \neq \emptyset$. Lemma A.2 shows that for $\Lambda_i, \Lambda_j \in \Psi$, if $\nu \in V(\Lambda_i), \nu \in V(\Lambda_j) \iff \Lambda_i = \Lambda_j$. In other words, no two chromons contain the same vertex. Since for any $\nu \in V(G)$ we know that there exists a $\Lambda \in \Psi$ (by definition of Ψ) with $\nu \in V(\Lambda)$. These Lemma's suggest that Ψ is indeed a partitioning of the vertex set $V(G)$.

□

Proposition A.4 (Uniqueness of Partition) *Let G be an arbitrary graph, C a set of colours, and $\gamma : V(G) \to C$. Let Ψ be the set defined in Defintion 2.4. Then Ψ is unique.*

Proof

Suppose that Ψ is not unique. Then there must be two partitions $\Psi_1 = \Psi_1(G, C, \gamma) \neq \Psi_2(G, C, \gamma) = \Psi_2$. Then we can find a vertex $\nu \in V(G)$ statisfying $\nu \in V(\Lambda_i), \nu \in V(\Lambda_j)$ for $\Lambda_i \in \Psi_1, \Lambda_j \in \Psi_2$ with $V(\Lambda_i) \neq (\Lambda_j) \implies \Lambda_i \neq \Lambda_j$.

We know that Ψ_1, Ψ_2 are produced by the same γ. So, by Lemma A.2, we must have $\Lambda_i = \Lambda_j$. This is a contradiction. So $\Psi_1 = \Psi_2$.

□

.

Proposition A.5 (Bounds on chromon size) *Let G be abitrary, with C a set of colours, and $\gamma : V(G) \to C$. Let Ψ denote the set defined in Definition 2.4. Let $|V(G)| = n$. Then for any $\Lambda \in \Psi$, $1 \leq |\Lambda| \leq n$.*

Proof

We know from Proposition A.3 that Ψ is a partition. From the definition of Ψ, we know that for any $\nu \in V(G)$, there exists a $\Lambda \in \Psi$ with $\nu \in V(\Lambda)$. It follows that:

$$\sum_{i=1}^{|\Psi|} |V(\Lambda_i)| = n$$

where $\Lambda_i \in \Psi$ and $\Lambda_i \neq \Lambda_j$ for $i \neq j$. Since we define $|\Lambda_i| = |V(\Lambda_i)|$:

$$\sum_{i=1}^{|\Psi|} |\Lambda_i| = n$$

It follows that $|\Lambda_i| \leq n$. It is also true that $|\Lambda_i| > 0$, since $\Lambda_i \neq \emptyset$. Since $|\Lambda_i| \in \mathbb{N}$, $|\Lambda_i| > 0 \implies |\Lambda_i| \geq 1$. Then,

$$1 \leq |\Lambda_i| \leq n$$

□

B Appendix - Notes on building the software

Building the software is done by simply issuing a `make` at the top level of the source tree. A `make clean` removes all object and binary files. The following is a description of each level of the source tree:

- `algorithm` - source code of all the algorithms.
- `exception` - exception handler sources.
- `generator` - random graph generator source code.
- `monograph` - source code of the abstract data structures.
- `program` - source code of the software interfaces.
- `testing` - some utlilities that were written for testing.
- `include` - the headers files for the different parts of the software - they are in appropriate subdirectories.
- `lib` - where the libraries are stored after they are built.

There are some parameters in the `Makefile` that can be adjusted to fine tune software compilation process. The first is the `USERFLAGS` variable. Generally, for debugging purposes, `USERFLAGS=-D__DEBUG__` is set. This causes the most critical parts of the software to be verbose. It should not be enabled for normal use.

Secondly the `CC` variable sets the compiler to be used. The code in the source tree uses some extensions by `gcc`. So, this variable should only be changed if a different version of `gcc` is to be used when compiling.

Lastly the `OPT` variable sets the optimisation flag to use. It is by default set to `OPT=-O2`. If the code is to be debugged with the GNU DeBugger, then `OPT=-ggdb3` should be set.

All other variables are only for internal use. It is not advisible that they should be altered.

C Appendix - Graph file format

The format for storing graphs allows graph files to be mainpulated using standard Unix™ text file commands. All graphs are separated by a ; symbol. An edge connecting two vertices `u` and `v` is given by `(u,v)`. From the last the example given in §5.1 graphs can be combined into a single file by the command: `cat 10000.0 10000.1 10000.2 10000.3 > 10000-12.graphs`.

For clarity, we an example of two graphs of size 6 stored in a single file

```
(0, 1)
(0, 2)
(0, 5)
(0, 4)
(1, 2)
(1, 3)
(1, 5)
(2, 3)
(2, 4)
(3, 5)
(3, 4)
(4, 5)
;
(0, 3)
(0, 4)
(0, 5)
(0, 1)
(1, 5)
(1, 2)
(1, 4)
(2, 4)
(2, 3)
(2, 5)
(3, 5)
(3, 4)
;
```

D Appendix - Data Types

D.1 Node

The Node type is an implementation of a vertex, defined in include/monograph/node.h:

```
typedef struct _Node            Node;
struct _Node {
        unsigned int    id;
        Colour          colour;

        unsigned int    neighbours;
        Node            *neighbour [DEGREE];

        NodeOps         *method;
};
```

The id field associated an identifier. The colour field associates a colour with the node and the neighbours field gives the degree of the node. The neighbour i can be accessed by dereferencing neighbour [i]. The recommended approach to manipulating a node is via the method variable which allows access to the methods defined on a Node. This is especially true for destructive updates (i.e., writing) than for reading.

We have a constructor and destructor for this type:

- Node *create_node (unsigned int i)
  Returns a pointer to an initialised node object, with default colour black, with id=i.
- void delete_node (Node *n)
  Deletes the object pointed to by n.

The following are the method defined on the Node structure. Note that the first argument of each method is a pointer to a Node object.

- unsigned int get_id (Node *n)
  Returns the id of the object n.
- void add_neighbour (Node *n, Node *neighbour)
  Adds the neighbour neighbour to n, if that neighbour doesn't already exist. It also maintains neighbours.
- Node *get_neighbour (Node *n, unsigned int i)
  Returns the neighbour in neighbour [i]. If there exists no such neighbour, NULL is returned.
- void del_neighbour (Node *n, Node *neighbour)
  If neighbour is a neighbour of n, then neighbour is removed from n's neighbours list.
- unsigned int neighbour_count (Node *n)
  Returns neighbours, i.e., the number of neighbours n has.
- Colour get_colour (Node *n)
  Returns the current colour of the node n.
- void set_colour (Node *n, Colour c)
  Changes the colour of n to c.
- void invert (Node *n)
  If the colour of n is black, then it is changed to white. If it is white, then it is changed to black.

- `void reset (Node *n)`
  This resets the status of `n` as if it were newly created. All neighbours are removed, and the colour is reset to `black`.

An example of access to a method would be: `n->method->invert (n);`.

D.2 MonoGraph

The `MonoGraph` Type is an implementation of a graph of degree 4 that can be coloured `black` or `white`. This type is defined in `include/monograph/graph.h`:

```
typedef struct _MonoGraph               MonoGraph;
struct _MonoGraph {
        unsigned int            vertices;

        Node                    **vlist;

        MonoGraphOps            *method;
};
```

The `vertices` field keeps track of the total size of the number of vertices, and `vlist [i]` allows access to the node with `id=i`.

Objects of this type should only be created via its constructor and destructor.

- `MonoGraph *create_graph (unsigned int v)`
  Create a graph of size `v`, and return a pointer to it.
- `void delete_graph (MonoGraph *g)`
  Delete the graph pointed to by `g`.

The field `method` defines the methods that are implemented for this type. It is recommended that these methods are used for read/write access. It is particularly not recommended to perform destructive updates outside of these methods. The following are the methods defined on this type:

- `Node **get_nodes (MonoGraph *g)`
  Returns the `vlist`.
- `Node *get_node (MonoGraph *g, unsigned int node)`
  Returns `vlist [node]`. That is, the node with `id=node`.
- `unsigned int get_size (MonoGraph *g)`
  Returns the size of `g`, that is `vertices`.
- `void connect (MonoGraph *g, unsigned int v, unsigned int u)`
  Adds the `u` node to the neighbours of `v` node. This should only be used to build directed graphs. If `v` cannot add another neighbour, then an exception is raised as a warning, but not to abort.
- `void uconnect (MonoGraph *g, unsigned int v, unsigned int u)`
  Works like the previous method. However, this also adds `v` as a neighbour of `u`. This is the way to build undirected graphs.
- `void disconnect (MonoGraph *g, unsigned int v, unsigned int u)`
  If `u` is a neighbour of `v`, then `u` is removed as a neighbour of `v`. Otherwise, nothing is performed. Calling this asymmetrically on an undirected graph causes it to become directed.

- `void reset (MonoGraph *g)`
  Returns the graph to a state as if it was newly created.
- `MonoGraph *duplicate (MonoGraph *g)`
  Duplicates the graph exactly, and returns a pointer to a copy.

D.3 List

The type `List` is an important type frequently used throughout. It is defined in `include/monograph/list.h`:

```
typedef struct _ListNode        ListNode;
struct _ListNode {
        void            *data;

        ListNode        *next;
        ListNode        *prev;
};

typedef struct _List            List;
struct _List {
        ListNode        *root;

        ListNode        *last;
        unsigned int    length;
        ListNode        *cur;

        ListOps         *method;
};
```

The `ListNode` data structure is an internal structure used as the container type to store the data. It allows for genericity by bypassing the typing mechanism, by making the `data` field of type `void *`. It is, therefore, the programmer's responsibility to ensure type consistency. The `next` and `prev` fields are used to link each the nodes appropriately in the list.

In the `List` structure, the field `root` is a pointer to the start of the list, and `last` to its end. The `cur` field is used for internal caching, to speed lookup and also to bookmark for methods that require "remembering" the previous position. The field `length` specifies the number of items in the list.

It is important to note that even the destructive methods described below do not modify the stored data in `data`. This implies that it is the programmer's responsibility to remove allocation of `data` to avoid memory leaks.

The `List` objects should be created and destroyed using the constructor and destructor provided:

- `List *create_list (void)`
  Returns a `List` object.
- `void delete_list (List *l)`
  Deletes the `List` object `l`.

The methods are accessed via the `method` field. The methods provided are described:

- `void *head (List *l)`
  Returns the element pointed to by `root`, and removes that element from the list. If the list is empty, then `NULL` is returned.

- `void *ndhead (List *l)`
  Like the previous operation, but does not remove the `root` element. Thus, consecutive calls will return the same `data`.
- `void *next (List *l)`
  Sets the `cur` field to `cur->next`, and returns the data in `cur`. If `cur` is `NULL`, then `cur` is set to `root`. If `cur` is `last`, then `NULL` is returned.
- `void *prev (List *l)`
  Sets `cur` to `cur->prev` and returns the data in `cur`. If `cur` is `NULL`, then `cur` is set to `last`. If `cur` is `root`, then `NULL` is returned.
- `void map (List *l, void (*mapfunc) (void *data, void *param), void *param)`
  Takes a function of type `void f (void *d, void *p)`, and applies it to all the elements of the list, where `d` is an item (ie `data`), and `p` is the `param` passed to this method.
- `ListNode *cons (List *l, void *data)`
  Adds `data` to the start of the list, and returns a pointer to the container holding `data`.
- `ListNode *append (List *l, void *data)`
  Like previous, but attaches to the end of the list.
- `List *merge (List *l, List *m)`
  Merges the elements of the list `m` into `l`. This destroys the object `m`.
- `unsigned int length (List *l)`
  Returns the length of the list `l`.
- `unsigned int nlength (ListNode *n)`
  Returns the length of the list from `n` to the end of the list.
- `void reset (List *l)`
  Sets `cur` to `NULL`.

Unlike the other data types, it should be emphasised that the list structure is optimised and complex. Any access (possibly with the exception of finding length) should be done strictly via the methods.

D.4 ChromonList

The `ChromonList` data type is optimised for maintaining the chromons of a graph. It is defined in `monograph/chromon.h`:

```
typedef struct _ChromonList     ChromonList;
struct _ChromonList {
        List            *black;
        List            *white;

        /* Keeps track of which chromon each node is in */
        ChromonV        *vertice;

        /* We need to access the graph */
        MonoGraph       *graph;

        /* Methods */
        ChromonOps      *method;
};
```

The `black` and `white` fields keep track of the chromons which are of colour black and white respectively. They are lists of lists. That is, a `data` field of an item in `black` itself is a list. But this list is a list of `Node` objects. The list specifies the vertices belonging to a chromon.

The `vertice` field keeps track of the chromon to which each vertex belongs to. The type `ChromonV` is simply a list. This allows $O(1)$ access to any chromon given a vertex. The `graph` field stores the graph for which the chromons are maintained.

Creation and destruction of `ChromonList` objects should be done via the constructor and destructor:

- `ChromonList *create_chromonlist (void)`
  Returns a `ChromonList` object.
- `void delete_chromonlist (ChromonList *c)` Destroys the object `c`.

The methods are accessed via `method`. They are:

- `void discover (ChromonList *c, MonoGraph *g)`
  Discovers the chromons of a graph `g`.
- `void rediscover (ChromonList *c, List *l)`
  Expects `l` a list of `Node` objects. It rediscovers the chromons to which each `Node` object belongs to in the `ChromonList` object `c`. It leaves the list `l` empty upon successful completion. If `discover` has not been called previously, it does nothing.

E Appendix - Graph Generator Implementation

The main data structure that the generator uses is defined in `include/generator/generator.h`:

```
typedef struct _Edge            Edge;
struct _Edge {
        unsigned int    u;
        unsigned int    v;

        unsigned int    r;
        unsigned int    interval;
};
```

In this data structure, `u` and `v` are the two ends of this pseudo-edge. The value `r` is $r_{\{u,v\}}$ defined in §4. The approach is to keep an array of size $\frac{n(n-1)}{2}$ of `Edge` structures. Then, at each instance, our value for `interval` is given by adding `r` to the `interval` of the previous index. For the index 0, our interval is simply the `r` value. When an edge is added, we move the pseudo-edge stored at the last index to the position of the added pseudo-edge, and decrement the size of our array accordingly. We then recompute the relevant `r` values, and recompute the `interval` values of the whole array.

This is a description of the functions which are used in the implementation of the graph generator:

- `MonoGraph *generate_graph (unsigned int vertices)`
  This is the caller function which calls other appropriate internal functions and returns a random 4-regular graph of size `vertices`. It ensures that the graph returned is 4-regular.
- `MonoGraph *generate_graph (unsigned int vertices, long int seed)`
  This is another caller function which calls `generate_graph()`, but it uses the value `seed` as a seed to the random number generator. If the `seed` given produces a graph that is not 4-regular, then the behaviour is changed to that of `generate_graph()` to avoid looping forever.
- `void initialise (void)`
  This function gets allocation for the pseudo-edge array, sets the correct value of the normalisation variable, and creates the array of possible edges.
- `void place (unsigned int i, unsigned int j)`
  Helper function, called by `initialise()` when it is creating the array of pseudo-edges. It takes care of placing all the appropriate values in each field of `Edge`.
- `void init_random (void)`
  This function seeds the random number generator by reading a `long int` from the random source `/dev/random`.
- `void make_graph (void)`
  This implements the main loop for generating the graph. It produces a random number, finds the pseudo-edge to which it belongs, and adds the edge to the graph. It uses a binary search on the array for efficiency.
- `void update_list (unsigned int index)`
  Helper function called by `make_graph()`, when it has added the pseudo-edge given by `index`. This function makes the array consistent again for the next run, updates the normalisation variable, appropriate `r` values and the `interval` of each pseudo-edge.

When a pseudo-edge $\mu\eta$ is added, we define the appropriate `r` value to mean any edge that has one end in one of the vertices μ or η. The graph generator implementation uses optimised rules for updating the normalisation variable and the appropriate `r` values.

F Appendix - Algorithm Implementations

F.1 Generic Functions

These are functions which are independent of the specific behaviour of a particular algorithm and are useful for all. They are defined in `monograph/generic.c`.

- `bool stable (Node *n)`
  This function takes a `Node` object, and returns `true` if it is stable, and `false` otherwise.
- `bool balanced (Node *n)`
  Returns `true` if the `Node` object `n` is 4-balanced, and `false` otherwise.
- `bool balanced2 (Node *n)`
  As previously, but for 2-balanced vertices.
- `gstabilise (MonoGraph *g)`
  This function gives an assignment to the vertices in `g` such that all vertices are balanced. On input, the vertices in `g` can be arbitrarily assigned a colour.
- `void print_chromonlist (ChromonList *c)`
  Function useful for debugging, as the name suggests. Given a `ChromonList` object `c`, it outputs it to `stdout` in a human readable form.
- `unsigned int cochromatic (Node *n)`
  Returns the number of co-chromatic neighbours that `n` has.
- `void stabilise (Node *n, List *fliplist)`
  Given that the all the vertices of graph to which `n` was stable, and `n` was flipped, it returns all possible affected vertices of `n` (and thus all vertices of the graph) back to a stable state. If `fliplist` is not `NULL`, then it adds all flipped vertices to this list.
- `void flip_and_stabilise (Node *n, List *fliplist)`
  Expects `n` to be balanced and the graph to which `n` belongs stable. It flips `n` and calls the `stabilise()` function. Then, the graph to which `n` belongs will still be stable but with `n` flipped.

F.2 EF Algorithm

We describe the (relevant) functions used in implementing the EF algorithm.

- `ChromonList *ef_algorithm (MonoGraph *g)`
  Caller function. Given a graph `g`, it runs EF on the graph and returns a `ChromonList` object.
- `void operationa (MonoGraph *g)`
  This function simply does a `gstabilise()` on the graph `g`.
- `void operationb (MonoGraph *g)`
  This expects all vertices in `g` to be stable. It creates a list of candidates for operation (b) and calls `opb()`. As a postcondition, we have a graph that is both stable and operation (b) cannot be done any further.
- `ChromonList *operationc (MonoGraph *g)`
  This takes a graph which is stable and with operation (b) complete. It builds a list of candidates for operation (c) and calls `opc()`. As a postcondition, we have a graph that has operation (a), (b), and (c) done such that none of these operations can be done any more.

- `void opb (List *wlist, List *opbflips)`
  This function expects the vertices in `wlist` as the candidates for operation (b). For each two vertices it flips, it performs a `stabilise()` to ensure that the all the vertices in the graph are stable. The appropriate vertices flipped from `stabilise()` are again looked at as candidates for a operation (b). This process is continued until there are no more candidate vertices - ie the operation cannot be done any further. If given `opbflips` not `NULL`, then all the flips (including those from `stabilise()`) are added to this list.

- `void opc (ChromonList *c, List *wlist)`
  This expects a ready-discovered `ChromonList` object `c`, and a list of candidates `wlist` on which to perform operation (c). Any vertex that it flips, a local `stabilise()` is performed, then from the list of all flipped vertices, it performs an `opb()`. This ensures that we've returned the graph to the state at the end of `operationb()`. Then the list of flipped vertices from `opb()`, and all cochromatic neighbours are then added to `wlist` as possible candidates for operation (c). The relevant chromons are rediscovered for consistency. This process is continued until there are no more vertices left to look at.

F.3 HST Algorithm

We describe the appropriate functions defined in implementing the HST algorithm.

- `ChromonList *hst_algorithm (MonoGraph *g)`
  Caller function. Given a graph g, it runs HST on the graph and returns a `ChromonList` object.

- `void whitify (MonoGraph *g)`
  This is an implementation of the `initialise_colouring()` phase of the algorithm as described in §3.4. However, it expects that all vertices of g are stable. It completely eliminates ripe vertices by performing the relevant operations, and then looks at white-balanced vertices. After flipping each white balanced vertices, it ensures that any ripe vertices created are once again eliminated. As a postcondition, it ensures that the graph g contains no ripe vertices or white balanced vertices.

- `List *get_blacks (MonoGraph *g)`
  This is an implementation of the `black_search()` functionality of the HST algorithm. It expects that all white chromons have size ≤ 2. It returns a list of vertices which are black, 4-balanced, and not adjacent to each other.

- `void deliberate_assignment (MonoGraph *g, List *blacks)`
  This is an implementation of the `assign_final_colouring()` functionality of the HST. It takes in a list `blacks` of black, 4-balanced, mutually non-adjacent vertices, and chooses which ones to colour white and black.

- `Node *ripe (Node *n)`
  This function is used by `whitify()` to determine whether a vertex is ripe. It returns a K_1 neighbour of `n` if `n` is ripe, and `NULL` otherwise.

- `bool wbalanced (Node *n)`
  Returns `true` if `n` is white and balanced, and `false` otherwise.

- `bool circuit (List *chromon)`
  Given `chromon`, and ordered list of vertices such that it defines a path or circuit, it returns `true` if the chromon is a path, and `false` otherwise.

- `Node *path_rewind (Node *n)`
  Given a node `n` which belongs to either a path or circuit, it picks an arbitrary end of a path as the start and returns that node. If `n` belongs to a circuit, it returns `n`.

- `bool distance (Node *n, Node *m, Colour c)`
  Given two vertices `n` and `m`, it returns true if there is a path across 2 or 3 vertices of colour `c` from `n` to `m`. Otherwise, it returns `false`.

- `void give_direction (Node *start, bool *done)`
  Given the start of a path or an arbitrary vertex in a circuit `start`, this gives the path or circuit a consistent direction. It also expects a `done` array to record what has already been done.

- `void do_assignments (MonoGraph *g, MonoGraph *h1, MonoGraph *h1p, MonoGraph *h2, MonoGraph *h2p, unsigned int *blacklist, unsigned length blacklength)`
  This function implements the `while` loop in the `assign_colouring()` phase of the HST algorithm as well as the colouring of white and black vertices. It expects `h1` and `h2`, successor graphs implementing $H[1]$ and $H[2]$ respectively. It also expects the graphs of predecessors - that is the opposite direction of $H[1]$ and $H[2]$, as `h1p`, and `h2p`. The `blacklist` array is a copy of the list, but it is also used for mapping forward and reverse from the `id` of nodes in `h1` and `h2` to the original graph `g`. The parameter `blacklength` gives the length of `blacklist`. This is the last functional phase of the HST algorithm.

- `unsigned int unplaced_successor (MonoGraph *h, unsigned int u, bool *done)`
  This expects `h` directed. It checks whether the neighbour `u`'s neighbour `v` is not recorded as `done`. If we have `v` done, then `h->vertices` is returned. Otherwise `v` is returned. This is often called with the graph defining predecessors to check for an unplaced predecessor. It does not update `done`.

F.4 EHST Algorithm

Most of the functions used in EHST are the same as that of HST. There are three functions different and two functions new.

- `ChromonList *ehst_algorithm (MonoGraph *g)`
  Caller function. Given a graph `g`, it runs EHST on the graph and returns a `ChromonList` object.

- `void ewhitify (MonoGraph *g)`
  This is an implementation of the `einitialise_colouring()` phase of the algorithm as described in §3.5. It looks at ripe vertices and white balanced vertices with equal priority. It is conjectured that it shares the same postcondition of `whitify()`.

- `void edeliberate_assignment (MonoGraph *g, List *blacks)`
  As before, but is an implementation of the `eassign_final_colouring()`. Rather than calling `do_assignments()`, it calls `edo_assignments()`.

- `void edo_assignments (MonoGraph *g, MonoGraph *h1, MonoGraph *h1p, MonoGraph *h2, MonoGraph *h2p, unsigned int *blacklist, unsigned length blacklength)`
  The behaviour is the same as the previous section, but implements the `while` loop in `eassign_final_colouring()`.

- `ChromonList *minimise_chromons (MonoGraph *g)`
  This take a graph `g` that has had `edeliberate_assignment()` performed as a last step. It builds a `ChromonList`, and flips a vertex if would make it belong to a smaller chromon. The postcondition is that the maximum chromon size is at most as large as from `edeliberate_assignment()`.

- `bool flipcheck (ChromonList *clist, Node *n)`
  Returns `true` if flipping `n` would make it belong to a smaller chromon, `false` otherwise.

G Appendix - Results Tables

The following key is used to interpret the data given in the table:

- $|V|$ - Number of vertices of graph.
- $\overline{M_c}$ - Average Maximum monochromatic component size.
- $\overline{A_c}$ - Average Average monochromatic component size.
- M_c - Maximum Maximum monochromatic component size.
- A_c - Maximum Average monochromatic component size.
- A_t - Average time.
- M_t - Maximum time.
- S_M - Standard deviation of Maximum monochromatic component size.
- S_A - Standard deviation of Average monochromatic component size.
- S_t - Standard deviation of time.

$\|V\|$	$\overline{M_c}$	$\overline{A_c}$	M_c	A_c	A_t	M_t	S_A	S_M	S_t
20	4.200000	1.937489	6	2.500000	40.000000	1000	0.276781	0.761052	39.979995
40	4.480000	1.887496	6	2.500000	39.959999	999	0.262261	0.809938	39.939995
60	4.800000	1.845866	6	2.142857	199.960007	1000	0.252011	0.867179	89.380107
80	5.000000	1.858072	6	2.222222	359.920013	1000	0.254475	0.903327	119.913330
100	5.040000	1.789242	6	2.083333	479.959991	1000	0.238965	0.912140	138.483223
200	5.840000	1.835193	9	2.083333	679.960022	1000	0.248688	1.085173	164.832051
300	5.800000	1.767177	8	1.898734	1119.880005	2000	0.233430	1.070327	233.114590
400	5.920000	1.765544	7	1.886792	1399.800049	2000	0.232934	1.089587	296.510391
500	6.240000	1.795206	8	1.960784	1919.680054	2999	0.239473	1.153776	391.755039
600	6.400000	1.779619	8	1.886792	2679.479980	3999	0.235869	1.185243	546.791406
700	6.360000	1.776912	8	1.871658	2919.600098	3000	0.235112	1.175755	586.336445
800	6.560000	1.784419	8	1.873536	3479.280029	4999	0.236785	1.215895	707.431484
900	6.640000	1.785633	9	1.923077	4039.320068	5000	0.237151	1.241934	814.625312
1000	6.360000	1.783734	7	1.838235	4919.200195	5999	0.236594	1.171665	990.094141
2000	6.800000	1.780329	8	1.828154	15957.719727	19997	0.235783	1.263646	1859.792656
3000	7.040000	1.775944	9	1.828154	34114.718750	43994	0.234830	1.314534	2450.545781
4000	7.560000	1.778671	9	1.811594	69749.476562	83988	0.235407	1.415345	2378.513437
5000	7.640000	1.778071	10	1.814224	129180.437500	160976	0.235265	1.435549	2590.210312
6000	8.160000	1.777822	12	1.821494	201729.406250	246962	0.235226	1.548160	1852.613437
7000	7.920000	1.780321	9	1.813472	306873.375000	363945	0.235753	1.484857	1745.621406
8000	7.920000	1.777615	11	1.809136	405298.437500	477927	0.235161	1.494523	2258.932500
9000	7.800000	1.779414	10	1.822600	544797.375000	629904	0.235550	1.466424	1832.853906
10000	8.040000	1.775365	9	1.804403	733288.625000	869868	0.234667	1.510497	1725.626094

EF Results

$\|V\|$	$\overline{M_c}$	$\overline{A_c}$	M_c	A_c	A_t	M_t	S_A	S_M	S_t
20	3.720000	1.659394	5	2.000000	199.960007	1000	0.212482	0.647457	89.380107
40	3.880000	1.565389	5	1.739130	440.000000	1000	0.189018	0.678822	132.598643
60	4.120000	1.551593	5	1.764706	319.920013	1000	0.186152	0.726636	113.052227
80	4.160000	1.543603	6	1.666667	519.919983	1000	0.183846	0.739730	144.127744
100	4.520000	1.526956	7	1.724138	719.880005	1000	0.180011	0.809938	169.592480
200	4.600000	1.527965	5	1.639344	1639.599976	2999	0.179848	0.819756	346.227734
300	5.040000	1.534089	7	1.595745	2559.679932	3999	0.181147	0.913893	524.427344
400	5.320000	1.519806	7	1.587302	3719.399902	5000	0.177852	0.969948	751.315703
500	5.360000	1.536096	7	1.612903	5159.240234	6999	0.181637	0.974885	1045.118984
600	5.400000	1.531592	7	1.578947	6079.160156	7999	0.180577	0.983056	1223.495312
700	5.360000	1.526067	7	1.583710	7238.959961	7999	0.179259	0.974885	1455.712656
800	5.400000	1.515751	7	1.581028	9478.839844	11999	0.176936	0.983056	1901.666250
900	5.280000	1.529187	7	1.578947	10758.360352	12998	0.179990	0.956661	2163.639531
1000	5.680000	1.522046	7	1.582278	13397.919922	14998	0.178368	1.043072	582.812305
2000	5.880000	1.522232	7	1.555210	46433.039062	53992	0.178349	1.082220	2122.617500
3000	5.960000	1.523521	8	1.547988	93305.882812	111983	0.178626	1.102724	1461.295000
4000	6.560000	1.521656	8	1.540832	177133.156250	218967	0.178197	1.215895	2550.576250
5000	6.640000	1.527783	8	1.552795	325510.468750	371944	0.179605	1.232883	1571.647813
6000	6.480000	1.520632	8	1.539251	491845.031250	570913	0.177961	1.201333	2407.787188
7000	6.720000	1.525447	8	1.538462	722010.187500	855870	0.179064	1.245793	2334.142500
8000	6.640000	1.524057	8	1.536098	953934.812500	1155824	0.178746	1.231584	1870.769063
9000	6.680000	1.521534	8	1.535836	1279725.500000	1469777	0.178168	1.236770	2401.533281
10000	6.960000	1.520874	8	1.535155	1693422.625000	1993697	0.178014	1.292440	1762.732344

HST Results

$\|V\|$	$\overline{M_c}$	$\overline{A_c}$	M_c	A_c	A_t	M_t	S_A	S_M	S_t
20	3.280000	1.711111	4	2.222222	199.919998	1000	0.226062	0.554256	89.362217
40	3.600000	1.599888	4	1.818182	359.959991	1000	0.197590	0.622254	119.926660
60	3.640000	1.631738	5	2.000000	559.919983	1000	0.205303	0.632456	149.570068
80	3.720000	1.570773	5	1.739130	759.880005	1000	0.190321	0.644981	174.241230
100	3.920000	1.590580	4	1.851852	919.840027	2000	0.194563	0.678822	199.875996
200	3.960000	1.565855	4	1.724138	1999.839966	3000	0.188756	0.685857	407.788203
300	4.120000	1.565490	5	1.639344	3319.439941	4000	0.188402	0.720000	672.693750
400	4.240000	1.551982	5	1.600000	4639.359863	5000	0.185217	0.746190	932.725937
500	4.280000	1.571242	5	1.677852	6119.080078	6999	0.189695	0.754718	1228.047578
600	4.280000	1.563941	5	1.675978	7278.959961	8999	0.188042	0.754718	1461.746875
700	4.240000	1.562792	5	1.605505	8918.519531	9999	0.187615	0.746190	1792.478125
800	4.440000	1.553313	5	1.626016	11118.200195	11998	0.185543	0.787909	2228.799219
900	4.400000	1.558792	5	1.624549	12798.120117	14998	0.186799	0.779744	2570.746250
1000	4.480000	1.549198	5	1.612903	15237.679688	16998	0.184585	0.795990	1567.611094
2000	4.680000	1.554741	5	1.602564	50312.281250	56991	0.185783	0.835225	2363.610625
3000	4.880000	1.556777	5	1.585624	105424.039062	124981	0.186221	0.872697	1289.711016
4000	4.960000	1.557919	5	1.581028	199449.718750	231965	0.186473	0.887243	1724.298281
5000	4.920000	1.556491	5	1.577287	353306.312500	387941	0.186154	0.880000	1941.385000
6000	5.000000	1.554941	6	1.573977	547756.812500	620906	0.185794	0.896214	825.233594
7000	4.960000	1.560363	5	1.582278	819675.250000	956854	0.187028	0.887243	1754.588438
8000	5.000000	1.559250	6	1.582278	1073636.750000	1233812	0.186774	0.896214	1538.008125
9000	5.000000	1.560353	6	1.577287	1452819.250000	1662747	0.187024	0.896214	566.850078
10000	5.040000	1.556098	6	1.568381	1947343.875000	2166671	0.186052	0.903327	1737.400156

EHST (LASHI) Results

References

[1] Edwards, K., Farr, G. *On monochromatic component size for improper colourings*, manuscript, October 2003

[2] Haxell, P., Szabo', T., Tardos, G. *Bounded size components - partitions and transversals*. J. Combinatorial Theory (Series B) **88** p. 281-297, 2003

[3] Steger, A., Wormald, N.C. *Generating random regular graphs quickly*. Combinatorics, Probability, Computing **8** p. 377-396, 1999

[4] Farr, G. *CSE3301: Algorithm descriptions*, manuscript, Wednesday March 17, 2004

[5] Farr, G. *CSE3301: Random regular graph generation*, manuscript, Wednesday April 7, 2004