UNIVERSITY
OF TRENTO

DEPARTMENT OF INFORMATION AND COMMUNICATION TECHNOLOGY

38050 Povo – Trento (Italy), Via Sommarive 14
http://www.dit.unitn.it

FINDING 1-FACTORS IN BIPARTITE REGULAR GRAPHS, AND EDGE-COLORING BIPARTITE GRAPHS

Romeo Rizzi

April 2002

Technical Report # DIT-02-0038

Finding 1-factors in bipartite regular graphs, and edge-coloring bipartite graphs

Romeo Rizzi*

April 25, 2002

Dipartimento di Informatica e Telecomunicazioni, Università di Trento
via Sommarive 14, 38050 Povo, Italy
romeo@science.unitn.it

Abstract

This paper gives a new and faster algorithm to find a 1-factor in a bipartite Δ-regular graph. The time complexity of this algorithm is $\mathcal{O}(n\Delta + n \log n \log \Delta)$, where n is the number of nodes. This implies an $\mathcal{O}(n \log n \log \Delta + m \log \Delta)$ algorithm to edge-color a bipartite graph with n nodes, m edges and maximum degree Δ.

Key words: time-tabling, edge-coloring, perfect matching, regular bipartite graphs.

1 Introduction

Let G be a bipartite regular graph. A celebrated result of Kőnig [5] (see [6] for a compact proof) states that G can be *factorized*, that is, $E(G)$ can be decomposed as the union of edge-disjoint 1-factors. (A 1-*factor* is simply another way to say perfect matching). Any bipartite matching algorithm can thus be employed to find a 1-factor in G and hence to factorize G. However, there exist faster methods exploiting the regularity of G. Cole and Hopcroft [1] gave an $\mathcal{O}(n\Delta + n \log n \log^2 \Delta)$ algorithm to find a 1-factor in a Δ-regular bipartite graph with n nodes. Schrijver [7] gave an $\mathcal{O}(n\Delta^2)$ algorithm for the same problem. Depending on the relative values of Δ and n, either algorithm gives the best-so-far proven worst-case asymptotic bound. We do not know of any randomized algorithm with better bounds.

In Section 2, we give an $\mathcal{O}(n\Delta + n \log n \log \Delta)$ deterministic algorithm, thus improving the bound on the side of Cole and Hopcroft's.

Let G be a bipartite graph (possibly not regular) with n nodes, m edges and maximum degree Δ. An edge-coloring of G assigns to each edge of G one of Δ possible colors so that no two adjacent edges receive the same color. By a simple reduction, the above cited result of Kőnig [5] implies that every bipartite graph admits an edge-coloring. Kapoor and Rizzi [4] gave an algorithm to edge-color G in $T_{n,m,\Delta} + \mathcal{O}(m \log \Delta)$ time, where $T_{n,m,\Delta}$ is the time needed to find a 1-factor in a d-regular bipartite graph with $\mathcal{O}(m)$ edges, $\mathcal{O}(n)$ nodes and

*Research carried out with financial support of the project TMR-DONET nr. ERB FMRX-CT98-0202 of the European Community and partially supported by a post-doc fellowship by the "Dipartimento di Matematica Pura ed Applicata" of the University of Padova

$d \leq \Delta$. Motivated by this result, we investigated Cole and Hopcroft's 1-factor algorithm for possible improvements. This effort culminated in the new and faster 1-factor procedure given in this paper. Combining this 1-factor procedure with the edge-coloring algorithm given in [4] we can edge-color G in $\mathcal{O}(n \log n \log \Delta + m \log \Delta)$ time.

2 The Algorithm

Our graphs have no loops but possibly have parallel edges. A graph without parallel edges is said to be *simple*. The *support* of a graph $\mathcal{G}$ is a simple graph G with $V(G) = V(\mathcal{G})$ and such that two nodes are adjacent in G if and only if they are adjacent in $\mathcal{G}$. The input of our algorithm is a bipartite Δ-regular graph $\mathcal{G}_0$ with n nodes and $m = \frac{n}{2}\Delta$ edges. We encode a graph $\mathcal{G}$ by giving its support G and by specifying for every edge uv of G the number $g[uv]$ of edges in $\mathcal{G}$ having u and v as endnodes. The number $g[uv]$ is a *positive* integer, called the *multiplicity* of edge uv in $\mathcal{G}$. Throughout the following, we should keep in mind that the proposed algorithms deal with graphs by actually manipulating supports and multiplicities.

In general, whenever $\mathcal{X}$ denotes a graph, then X stands for the support of $\mathcal{X}$ and x for the multiplicities' vector of $\mathcal{X}$. Even if no value $x[uv] = 0$ is stored explicitly by the algorithm, we will consider $x[uv]$ to be 0 when u and v are not adjacent in $\mathcal{X}$. All graphs considered are restricted to have the same node set V, namely $V = V(\mathcal{G}_0)$. The *sum* $\mathcal{G} + \mathcal{H}$ of two graphs $\mathcal{G}$ and $\mathcal{H}$ is the graph $\mathcal{S}$ with $s = g + h$ (componentwise). The maximum degree of a node in a graph $\mathcal{H}$ is denoted by $\Delta(\mathcal{H})$. Throughout the whole algorithm the value Δ will also be a constant and stands for $\Delta(\mathcal{G}_0)$.

We say that graph $\mathcal{G}$ *contains* graph $\mathcal{H}$ when $E(H) \subseteq E(G)$. When $\mathcal{G}$ contains $\mathcal{H}$ (in short $\mathcal{H} \subseteq \mathcal{G}$) and $\mathcal{H}$ contains a 1-factor then $\mathcal{G}$ also contains a 1-factor. Our algorithm will modify the input graph $\mathcal{G}_0$ thus determining a sequence $\mathcal{G}_0, \mathcal{G}_1, \ldots$ of graphs. Each graph in the sequence will be contained in the previous one and all graphs will be regular. The support of the last graph in the sequence will be a 1-factor.

A graph $\mathcal{G}$ is said to be *sparse* if $|E(G)| \leq 2n \log \Delta$. For our manipulations to be performed efficiently it will be crucial to assume we are working on sparse graphs. Thus a first phase of our algorithm will have to make $\mathcal{G}_0$ sparse. Subsection 2.1 describes a preprocessing algorithm to sparsify $\mathcal{G}_0$. This preprocessing algorithm was first proposed by Cole and Hopcroft in [1]. Here we prefer to describe it in some more detail.

2.1 Why we assume $\mathcal{G}_0$ to be sparse: the preprocessing phase

Cole and Hopcroft [1] proposed the following method to obtain a sparse Δ-regular graph $\mathcal{H}$ contained in a Δ-regular graph $\mathcal{G}$. The method takes $\mathcal{O}(m)$ time.

Obviously $g[e] \leq \Delta$ for every $e \in E(G)$. Let $k = \lfloor \log \Delta \rfloor + 1$ and let $g[e]_{[k]}, \ldots, g[e]_{[1]}, g[e]_{[0]}$ be the binary encoding of $g[e]$. This means that $g[e] = \sum_{i=0}^{k} g[e]_{[i]} \cdot 2^i$. For $i = 0, 1, \ldots, k$ define the edge-set

$$E_i(\mathcal{G}) = \{e \in E(G) \ : g[e]_{[i]} = 1\}$$

For example, $E_0(\mathcal{G})$ is the set of edges having odd multiplicity in $\mathcal{G}$.

Start with $\mathcal{H} = \mathcal{G}$. When each $E_i(\mathcal{H})$ is acyclic, then $|E_i(\mathcal{H})| < n$ for $i = 1, \ldots, k$, hence $\mathcal{H}$ is sparse. The idea is to first make $E_0(\mathcal{H})$ acyclic, then $E_1(\mathcal{H})$, and so on, until $E_k(\mathcal{H})$. Let C

be a cycle contained in $E_{\bar\imath}(\mathcal{H})$ with $\bar\imath$ as small as possible. Let M_1, M_2 be two matchings such that $C = M_1 \cup M_2$. Then by setting $h[e] \leftarrow h[e] - 2^{\bar\imath}$ for every edge e in M_1 and $h[e] \leftarrow h[e] + 2^{\bar\imath}$ for every edge e in M_2 we do not affect any of $E_0(\mathcal{H}), E_1(\mathcal{H}), \ldots, E_{\bar\imath-1}(\mathcal{H})$ but reduce $|E_{\bar\imath}(\mathcal{H})|$ by $|C|$. Note that this manipulation preserves the Δ-regularity of $\mathcal{H}$. Moreover the graph produced by the manipulation will be contained in the one it has been obtained from. This preprocessing algorithm can be implemented to run in time $\mathcal{O}\left(m + \frac{m}{2} + \frac{m}{4} + \ldots\right) = \mathcal{O}(m)$. We close this subsection with two more implementational subtleties.

1. After setting $h[e] \leftarrow h[e] - 2^{\bar\imath}$ we check if $h[e] < 2^{\bar\imath}$. If this is the case then $e \notin E_j$ for any $j > \bar\imath$ and edge e is removed from the "working input graph" and is placed in the "definitive graph". The "definitive graph" is output when the procedure terminates.

2. The search for circuit C is done as follows. Starting from a node v_o construct a depth-first search tree T and when a circuit C is detected, then all nodes of the tree but not in C which have a node of C as ancestor are guaranteed not to belong to any circuit in $E_{\bar\imath}(\mathcal{H})$, so we discard them and free the nodes in $V(C)$ after performing the above described manipulation. All the other nodes remain in the tree. When T is completed then we can discard all nodes in $V(T)$ and construct a new depth-first search tree starting from any (not-yet-discarded) node. When no node is left, then $E_{\bar\imath}$ is acyclic.

2.2 Why we assume Δ to be odd: Procedure *EulerSplit*

The reduction given in this subsection dates back to Gabow [2].

A graph $\mathcal{G}$ is called *Eulerian* when every node has even degree in $\mathcal{G}$. We first describe a basic procedure, called *EulerSplit*, which, given as input an Eulerian graph $\mathcal{G}$, returns a graph $\mathcal{H}$ with $h \leq g$ (componentwise) and such that for every node $v \in V$ the degree of v in $\mathcal{G}$ is twice the degree of v in $\mathcal{H}$. From the following description, Procedure *EulerSplit* can be implemented as to take $\mathcal{O}(n \log \Delta)$ time, when $\mathcal{G}$ is sparse.

Decompose G as $G_e + G_o$, where G_o contains precisely those edges of G which have odd multiplicity in $\mathcal{G}$. Since $\mathcal{G}$ is Eulerian, then G_o is Eulerian. By orienting the edges of G_o in the direction they are traversed by an Euler tour we find an orientation of G_o such that the in-degree equals the out-degree for every node. Now we decompose G_o as $\overleftarrow{G_o} + \overrightarrow{G_o}$, where $\overrightarrow{G_o}$ contains precisely those edges of G_o which have been oriented as to go from, let say, the "left" side of the bipartition to the "right" side. Consider the graph $\mathcal{H}$ contained in $\mathcal{G}$ and such that

$$h[e] = \frac{g[e]}{2} \quad \text{if } e \text{ is an edge of } G_e \qquad\qquad \begin{cases} h[e] = \left\lfloor \frac{g[e]}{2} \right\rfloor & \text{if } e \text{ is an edge of } \overleftarrow{G_o} \\ h[e] = \left\lceil \frac{g[e]}{2} \right\rceil & \text{if } e \text{ is an edge of } \overrightarrow{G_o} \end{cases}$$

Note that $h \leq g$ and for every node $v \in V$ the degree of v in $\mathcal{G}$ is twice the degree of v in $\mathcal{H}$.

The reason why we can always assume Δ to be odd is the following procedure.

Procedure 1 MAKEODD ($\mathcal{G}$) (precondition: $\mathcal{G}$ is regular)

1. if $\Delta(\mathcal{G})$ is odd then return $\mathcal{G}$;
2. else return $MakeOdd(EulerSplit(\mathcal{G}))$.

2.3 Procedure *Split* and taking complements

Our algorithm calls Procedure *Split*, an important operation due to Cole and Hopcroft [1].

A graph $\mathcal{S}$ is a *slice* of a graph $\mathcal{G}$ when $s \leq g$. Slice $\mathcal{S}$ is *big* when $|E(\mathcal{G})| \leq 2|E(\mathcal{S})|$. For $k \geq 1$, slice $\mathcal{S}$ is a $(k, k+1)$-*slice* if each node $v \in V$ has degree either k or $k+1$ in $\mathcal{S}$. We denote by $odd(\mathcal{S})$ the set of those nodes having odd degree in $\mathcal{S}$. The *complement* of a $(k, k+1)$-slice $\mathcal{S}$ in $\mathcal{G}$ is the unique graph $\mathcal{T}$ such that $\mathcal{S} + \mathcal{T} = \mathcal{G}$. Note that $\mathcal{T}$ is a $(\Delta - k - 1, \Delta - k)$-slice. Moreover, when Δ is odd, then $odd(\mathcal{T}) = V \setminus odd(\mathcal{S})$. When $\mathcal{G}$ is sparse, the complement can be computed in $\mathcal{O}(n \log \Delta)$ time.

Procedure *Split* takes as input a $(k, k+1)$-slice $\mathcal{S}$ of $\mathcal{G}$ and returns an $(h, h+1)$-slice $\mathcal{S}'$ of $\mathcal{G}$ with $|odd(\mathcal{S}')| \leq \frac{|odd(\mathcal{S})|}{2}$. The computation of $\mathcal{S}' = Split(\mathcal{S}; \mathcal{G})$ is accomplished as follows. Decompose S as $S_e + S_o$, where S_o contains precisely those edges of S which have odd multiplicity in $\mathcal{S}$. Orient the edges of S_o so that for every node the in-degree differs from the out-degree by at most 1. When $\mathcal{G}$ is sparse, this can be done in $\mathcal{O}(n \log \Delta)$ time by for example adding some artificial edges to S_o as to make it Eulerian and then proceeding as in Subsection 2.2. Decompose S_o as $\overleftarrow{S}_o + \overrightarrow{S}_o$ as explained in Subsection 2.2. Let w be the odd value in $\{k, k+1\}$. If $w = k$ then let $S_o{}^{up}$ be a big slice of S_o in $\{\overleftarrow{S}_o, \overrightarrow{S}_o\}$ and let $S_o{}^{down}$ be the other slice. Otherwise let $S_o{}^{down}$ be a big slice of S_o in $\{\overleftarrow{S}_o, \overrightarrow{S}_o\}$ and let $S_o{}^{up}$ be the other slice. Consider the graph $\mathcal{P}$ contained in $\mathcal{S}$ and such that

$$p[e] = \tfrac{s[e]}{2} \quad \text{if } e \text{ is an edge of } S_e \qquad \begin{cases} p[e] = \left\lceil \frac{s[e]}{2} \right\rceil & \text{if } e \text{ is an edge of } S_o{}^{up} \\ p[e] = \left\lfloor \frac{s[e]}{2} \right\rfloor & \text{if } e \text{ is an edge of } S_o{}^{down} \end{cases}$$

If $w = k+1$ then $\mathcal{P}$ is a $(\frac{k}{2}, \frac{k}{2}+1)$-slice where at most $\frac{|odd(\mathcal{S})|}{2}$ nodes have degree $\frac{k}{2}+1$. Therefore, if $\frac{k}{2}+1$ is odd then $\mathcal{S}' = \mathcal{P}$ will work and otherwise we will take as $\mathcal{S}'$ the complement of $\mathcal{P}$. If $w = k$ then $\mathcal{P}$ is a $(\frac{k+1}{2} - 1, \frac{k+1}{2})$-slice where at most $\frac{|odd(\mathcal{S})|}{2}$ nodes have degree $\frac{k+1}{2}$. Therefore, if $\frac{k+1}{2}$ is odd then $\mathcal{S}' = \mathcal{P}$ will work and otherwise we will take as $\mathcal{S}'$ the complement of $\mathcal{P}$. Note that, when $\mathcal{G}$ is sparse, then *Split* requires $\mathcal{O}(n \log \Delta)$ time.

2.4 The algorithm of Cole and Hopcroft

The following pseudo-code describes a simplified version[1] of Cole and Hopcroft's algorithm [1].

Algorithm 2 COLE_HOPCROFT ($\mathcal{G}_0$) (precondition: $\mathcal{G}_0$ is Δ-regular)

1. $\mathcal{G} \leftarrow MakeOdd(\mathcal{G}_0)$;
2. while G is not a 1-factor *invariant:* $\mathcal{G} \subseteq \mathcal{G}_0$ is regular with $\Delta(\mathcal{G})$ odd
3. $\mathcal{S} \leftarrow \mathcal{G}$;
4.   do $\mathcal{S} \leftarrow Split(\mathcal{S}; \mathcal{G})$;
5.   while $odd(\mathcal{S})$ is not empty; *invariant*[2]*:* $\mathcal{S}$ is a $(k, k+1)$-slice of $\mathcal{G}$
6. $\mathcal{G} \leftarrow MakeOdd(\mathcal{S})$;
7. return G.

[1] in the original version step 6. assigns to $\mathcal{G}$ the complement of $\mathcal{S}$ in $\mathcal{G}$, in case $\mathcal{S}$ is a big slice of $\mathcal{G}$.

Loop 4–5, when entered, cycles $\mathcal{O}(\log n)$ times, since $odd(\mathcal{S})$ is at least halved each time. Loop 2–6, when entered, cycles $\mathcal{O}(\log \Delta)$ times, since $\Delta(\mathcal{G})$ is at least halved each time. All operations involved in loop 2–6, except *MakeOdd*, cost $\mathcal{O}(n \log \Delta)$, since by Section 2.1 we can assume that $\mathcal{G}_0$ is sparse. Since *EulerSplit* is executed $\mathcal{O}(\log \Delta)$ times, the total time spent in *MakeOdd* over the whole execution of the algorithm is $\mathcal{O}(n \log^2 \Delta)$. Hence Cole and Hopcroft's algorithm is $\mathcal{O}(n\Delta + n \log n \log^2 \Delta)$.

2.5 Our starting point: Procedure *Starter*

Our starting point is essentially the inner loop in Cole and Hopcroft's algorithm. We have just shown its cost to be $\mathcal{O}(n \log n \log \Delta)$ for sparse input graphs. Here we assume Δ to be odd.

Procedure 3 STARTER ($\mathcal{G}$) (precondition: $\mathcal{G}$ is Δ-regular and Δ is odd)

1. $\mathcal{S} \leftarrow \mathcal{G}$;
2. do $\mathcal{S} \leftarrow Split(\mathcal{S}; \mathcal{G})$;
3. while $odd(\mathcal{S})$ is not empty; *invariant*[2]*:* $\mathcal{S}$ is a $(k, k+1)$-slice of $\mathcal{G}$
4. return $\mathcal{S}$.

The output $\mathcal{S}$ of Procedure *Starter* is a δ-regular graph contained in $\mathcal{G}$. A crucial property about $\mathcal{S}$ and $\mathcal{G}$ is that δ and Δ are *coprime*, that is, the only integer which divides both is 1. Indeed, regarding $\mathcal{G}$ as a $(\Delta - 1, \Delta)$-slice of $\mathcal{G}$, then $\mathcal{S} = Split(\mathcal{G}; \mathcal{G})$ is a $(\frac{\Delta-1}{2}, \frac{\Delta+1}{2})$-slice of $\mathcal{G}$, that is, a $(k, k+1)$-slice where both k and $k+1$ are coprime with Δ. A second invariant[2] of loop 2–3 in Procedure *Starter* is that the even value among k and $k+1$ is coprime with Δ. In fact, $g.c.d.(a, b) = g.c.d.(a, a-b)$ (taking complement) and $g.c.d.(a, 2b) = g.c.d.(a, b)$, assuming that a is odd.

The next subsection describes an algorithm, which given as input a Δ-regular graph $\mathcal{G}$ and a δ-regular graph $\mathcal{S}$, returns a regular graph $\mathcal{F}$ with $f \leq g + s$ and $\Delta(\mathcal{F}) = g.c.d.(\Delta; \delta)$ in $\mathcal{O}((|E(G)| + |E(S)|) \log^2 \Delta)$ time. In our case $s \leq g$ and $g.c.d.(\Delta, \delta) = 1$, hence a 1-factor of $\mathcal{G}$ is returned. Moreover $|E(S)| < |E(G)| = \mathcal{O}(n \log \Delta)$ and the time bound is $\mathcal{O}(n \log^3 \Delta)$. This term is dominated by the $\mathcal{O}(m)$ cost of the preprocessing phase.

2.6 Computing the $g.c.d.$ by sums and shiftings

When a and b are two positive integers we denote by $g.c.d.(a, b)$ the greatest common divisor of a and b. When both a and b are even then $g.c.d.(a, b) = 2\, g.c.d.\left(\frac{a}{2}, \frac{b}{2}\right)$. This section considers an algorithm to compute $g.c.d.(a, b)$ when at least one of a and b is odd. The procedure is allowed to use the following operations: dividing an even by 2 (this corresponds to *EulerSplit* and costs $\mathcal{O}(n \log \Delta)$), testing evenness, summing two integers (the sum of two graphs also costs $\mathcal{O}(n \log \Delta)$), and comparing two integers (greater, less, or equal?). The procedure goes as follows: When one of the two numbers is even then we divide it by 2 and the g.c.d. does not change since the other number is odd. So both numbers are odd.

[2] *second invariant:* Δ is coprime with the even value among k and $k+1$.

Therefore their sum σ is even and if we substitute the biggest of the two numbers by $\frac{\sigma}{2}$ the g.c.d. does not change. Eventually the two numbers will be equal. But now $g.c.d.(a,a) = a$.

We now show that the above procedure[3] uses $\mathcal{O}(\log^2(a+b))$ operations. This is because each time $\frac{\sigma}{2}$ is even then σ actually decreases at least by a factor of $\frac{3}{4}$, and when $\frac{\sigma}{2}$ is odd then $|b-a|$ decreases at least by a factor of 2, while σ is never increased.

Here is the algorithm promised in the end of the previous subsection:

Algorithm 4 G.C.D. $(\mathcal{G}, \mathcal{S})$ (precondition: $\mathcal{G}$ and $\mathcal{S}$ are regular)

1. $\mathcal{G} \leftarrow MakeOdd(\mathcal{G})$; $\mathcal{S} \leftarrow MakeOdd(\mathcal{S})$;
2. while $\Delta(\mathcal{G}) \neq \Delta(\mathcal{S})$
3. by eventually exchanging $\mathcal{G}$ and $\mathcal{S}$, assume $\Delta(\mathcal{G}) \geq \Delta(\mathcal{S})$;
4. $\mathcal{G} \leftarrow MakeOdd(\mathcal{G} + \mathcal{S})$.

Acknowledgments

I thank the referee for the detailed feedback he has given.

References

[1] R. Cole, J. Hopcroft, On edge coloring bipartite graphs, *SIAM Journal on Computing* 11 (1982) 540-546.

[2] H.N. Gabow, Using Euler partitions to edge color bipartite multigraphs, *International J. Computer and Information Sciences* 5 (1976) 345-355.

[3] H.N. Gabow, O. Kariv, Algorithms for edge coloring bipartite graphs and multigraphs, *SIAM Journal on Computing* 11 (1982) 117-129.

[4] A. Kapoor and R. Rizzi, Edge-coloring bipartite graphs, *Journal of Algorithms* 34 (2) (2000) 390–396.

[5] D. Kőnig, Graphok és alkalmazásuk a determinánsok és a halmazok elméletére [Hungarian], *Mathematikai és Természettudományi Értesito* 34 (1916) 104-119.

[6] R. Rizzi, Kőnig's Edge Coloring Theorem without augmenting paths, *Journal of Graph Theory* 29 (1998) 87.

[7] A. Schrijver, Bipartite edge-colouring in $\mathcal{O}(\Delta m)$ time, SIAM Journal on Computing 28 (3) (1999) 841-846.

[3] a deeper analysis of a related and similar procedure is given in [4]