A bipartite graph $H = (A, B, E)$ with $|A| = |B|$ that is k-complete with respect to r can be transformed to one that is k-filled with respect to r by a construction sketched as follows. Let $|A| = |B| = n$. Augment H so that $H' = (A \cup C, B \cup D, E \cup F)$ with vertex sets $|C| = |D| = n - k$ and edge set $F = AD \cup BC$ where $AD \subseteq A \times D$ and $BC \subseteq B \times C$. The edges in F are added and their weights assigned such that all vertices in H' have weight r and yet H' is also k-complete with respect to r. We call this algorithm k-**fill**. See Bongiovanni et al [10] for a detailed description of this algorithm and its proof of correctness. We observe that k-**fill** takes time $O(n)$.

Theorem 3.1 [10]. *For any bipartite graph $H = (A, B, E)$ with $|A| = |B|$ and bound q that is k-complete with respect to q, there exists a critical k-matching.*

proof. The proof is by construction, as outlined in the algorithm below.

Algorithm CKM(H, k, q)

Input: a bipartite graph H, which is k-complete with respect to its bound q

Output: A critical k-matching on H.

```
1. H' = $k$-fill(H);
2. Find a matching M on H'
3. return(M $\cap$ E)
4. end
```

Use k-**fill** to generate $H' = (A \cup C, B \cup D, E \cup F)$ from H. From Lemma 3.5, there exists a matching M of size $2n - k$ for H'. Since H' contains no edges in $C \times D$. $n-k$ edges of M are required to cover the vertices in C, and similarly for D. This leaves $2n-k-2(n-k)=k$ edges of M having vertices only in H. Further, these k edges cover every critical weight vertex in H, since such vertices do not need to be augmented by adding edges in F. Thus $M \cap E$ gives the required critical k-matching on H. □

We now cite the theorem that, together with Theorem 3.1, ensures that it is always possible to k-color G.

Def. The *duration* of a matching M of a graph with bound q is $r = \min(r', q-r'')$, where r' is the minimum of the edge weights of M and r'' is the maximum vertex weight among vertices not covered by M.

Theorem 3.2 *Let $H = (A, B, E)$ be a k-complete bipartite graph with $|A| = |B|$ which is k-complete with respect to its bound q. Let M be a critical k-matching on H. Let M' be M with the weight of all edges set to the duration r of M. Then the graph $HM = (A, B, E - M')$ is k-complete with respect to $q' = q - r$.*

proof. See Bongiovanni et al [10]. □

Informally, the duration of a matching is the number of time slots a critical k-matching can be used repeatedly. At each time slot that it is used, the weight of all the edges in the graph that are also in the matching decreases by one, since the remaining length of the transfer represented by that edge decreases by one. A new critical k-matching must be calculated if either the weight of one of the edges of the matching decreases to zero, or a vertex that was previously not critical weight becomes critical.

Theorems 3.1 and 3.2 together lead to an optimal algorithm called **KT** [10]. In **KT**, dummy traffic is added to augment the bipartite graph and make it k-complete with respect to its bound. The well-known max-min bipartite weighted matching algorithm (see [92]), which we call **MaxMinMatch**, is then invoked to find a critical k-matching. The result of Theorem 3.2 is used to calculate its duration. A sequence of critical k-matchings is found by calling the max-min bipartite algorithm repeatedly. Finally, the dummy traffic is removed from the k-matchings to obtain an optimal schedule.

It should be pointed out that the objective of Bongiovanni et al [10] was to obtain a minimum-length schedule while not paying too high a price in terms of L, the number of times that critical k-matchings have to be calculated. The concern in this paper is to obtain a minimum-length schedule while reducing the total running time of the algorithm. In the following we evaluate the running time of **KT** and suggest an improvement.

Theorem 3.3 *The running time of algorithm* **KT** *to find a minimum-length schedule for an instance of DTS is* $O(n^5)$.

proof. Augmenting a bipartite graph to make it k-complete is $O(n)$ (Lemma 3.4), as is deleting the dummy traffic. Bongiovanni et al [10] show that

$L = O(n^2)$, but the running time of **KT** is not calculated. However, we thus know that the number of times that the max-min bipartite matching algorithm **MaxMinMatch** [92] is called is $O(n^2)$. Since the max-min matching algorithm has a running time of $O(n^3)$, the result follows. □

3.4 An improved scheduling algorithm (A1)

In this section we show how **KT** can be improved. We recall that in order to minimize the running time, an algorithm that simply finds a maximum-cardinality matching in H', the regular weight bipartite graph with $2n - k$ vertices in each partition, suffices. Such an algorithm will find the required critical k-matching in H. We can thus use the maximum cardinality matching algorithm of [71], which we call **MaxMatch**. We call the resulting optimal scheduling algorithm **A1**.

Theorem 3.4 *The running time of algorithm* **A1** *to find a minimum-length schedule for an instance of DTS is* $O(n^{4.5})$.

proof. The maximum cardinality matching algorithm **MaxMatch** of [71] takes time $O(|V|^{.5}|E|) = O(|V|^{2.5})$ for a bipartite graph with $|V|$ vertices and $|E|$ edges. For **A1**, $|V| = 2n - k$. Using the maximum cardinality matching algorithm does not affect the worst-case value of L, or the time complexity of adding and deleting dummy traffic. We can use the reasoning of Theorem 3.3 to obtain the result. □

3.5 A divide-and-conquer scheduling algorithm (A2)

In this section we obtain an optimal algorithm for *DTS* which we call **A2**. The key observation is that the bipartite graph G can be partitioned into two graphs of roughly equal weight which represent two independent sub-problems of roughly half the complexity of the original problem. Then algorithm **A1** can be applied recursively to the subproblems to obtain an optimal schedule.

Def. A *walk* is a sequence of distinct adjacent edges. The first and last vertex of the sequence are called the *ends* of the walk. An *open* walk is one in which the ends are distinct; otherwise the walk is *closed*.

A walk differs from a path in that any vertex may be included more than once (not just the first vertex). This definition is used in the following two definitions, which are based on Cole and Hopcroft [29].

Def. An *Euler partition* of a graph is a partition of the edges into open and closed walks, so that each vertex of odd degree is at the end of exactly one open walk, and each vertex of even degree is at the end of no open walk.

Def. An *Euler split* of a bipartite graph $G = (A, B, E)$ is a pair of bipartite graphs $H = (A, B, F)$ and $H' = (A, B, F')$ where $E = F \cup F'$ and a vertex of degree d in G has degree $\lceil d/2 \rceil$ in one of H, H' and $\lfloor d/2 \rfloor$ in the other.

Every graph has an Euler partition, but only bipartite graphs need have Euler splits [5, 29]. An Euler split can be formed from an Euler partition of

G by placing alternate edges of walks into F and F'; both can be found in time $O(n+m)$ for bipartite graphs and multigraphs [48]. An Euler split of the multigraph G' corresponding to G, using the algorithm of Gabow [48], suffices to divide G into two subgraphs each having roughly half the weight of the original. However, this approach would take time proportional to the maximum edge weight in G. In the following we develop an algorithm which is faster because it avoids converting G into a multigraph.

Def. An *Euler division* of a weighted bipartite graph $G = (A, B, E)$ which is k-complete with respect to its bound q is a pair of bipartite graphs $H = (A, B, F)$ and $H' = (A, B, F')$ with $E = F \cup F'$ where the bounds of H and H' are $r = \lceil q/2 \rceil$ and $r' = \lfloor q/2 \rfloor$ respectively, and H, H' are k-complete with respect to their bounds.

In the following we will show that if q is even, an Euler division always exists. (This result is required for the divide-and-conquer algorithm **A2** we develop later in this section). The proof is constructive, and is based on the Euler partition algorithm of Gabow [48] and the *perfect Euler split* algorithm of Somalwar [132], both of which are designed for bipartite graphs with unit-weight edges. We use the notation $wt(Y, y)$ to refer to the weight of a vertex or an edge y in a weighted bipartite graph Y.

Algorithm ED

/* Euler Division of bipartite graphs with even bounds */

Input: Bipartite graph $G = (A, B, E)$ that is k-complete with respect to its bound q, which is even.

Output: An Euler division of G into bipartite graphs $H = (A, B, F)$, $H' = (A, B, F')$.

```
1. F, F' := {}, {};
2. for each e ∈ E such that wt(G, e) > 1 do
3.     wt(H, e), wt(H', e), wt(G, e) := ⌊ wt(G, e) / 2 ⌋, ⌊ wt(G, e) / 2 ⌋,
           wt(G, e) - 2 * ⌊ wt(G, e) / 2 ⌋;
4.     F, F' := F ∪ {e}, F' ∪ {e};
5. end for
6. G' := G;            /* G' introduced for convenience only */
7. P := EP(G');            /* EP generates an Euler partition [48]*/
8. balance := true;
9. for each walk p ∈ P do
10.     for each edge e ∈ p do
11.         if balance = true then
12.             wt(H, e), F := wt(H, e) + 1, F ∪ {e};
13.         else
14.             wt(H', e), F' := wt(H', e) + 1, F' ∪ {e};
15.         end if
16.         balance := ¬balance;
17.     end for
18. end for
```

Lemma 3.6 *Algorithm* **ED** *generates an Euler division of G.*

proof. The first **for** loop divides all even weight edges from G and converts all odd weight edges to unit weight. Thus G' is a unit-weight bipartite graph, and at line 6 H and H' have equal weights and equal degrees at every vertex. Algorithm **EP** generates an Euler partition for a unit-weight bipartite graph [48]. It is clear that lines 9 - 18 ensure that edges from walks in the Euler partition are assigned to H and H' in such a manner that the weights of H and H' differ by at most 1 at the end of the algorithm.

We now show that H, H' are an Euler division of G. Clearly $E = F \cup F'$ by construction. Let X and X' be, respectively, the weights of H and H', x, x' the weights of their critical vertices, and r, r' their bounds. Since q is even, and $W = kq$, W is even. Also, $W = X + X'$ and $|X - X'| \leq 1$ by construction. Hence $X = X' = kq/2$. Consequently, if we show that $r = r' = q/2$, then H, H' will be k-complete with respect to their bounds and thus be an Euler division of G.

By definition of bound, showing $r = r' = q/2$ requires that we show that x, $x' \leq q/2$. Let $z = wt(G, c)$ be the weight of an arbitrary vertex c in G at the start of the algorithm. Note that at line 1, if z is even, there are an even number of odd-weight edges incident upon c in G, and an odd number otherwise. Consequently, after line 6, $wt(G', c) = d$ is even if z is even, and odd otherwise. In either case, after line 6, $wt(H, c) = wt(H', c) = (z - d)/2$ is even. Next, recall that by definition an Euler partition results in an odd-degree vertex being at the end of exactly one open walk; hence observe that **EP** followed by lines 8 - 18 results in a vertex of weight d in G' contibuting weight at most $\lceil d/2 \rceil$ in H, H'. Therefore, lines 7 - 18 result in $wt(H,c), wt(H',c) \leq (z-d)/2 + \lceil d/2 \rceil = \lceil z/2 \rceil$. Since G is k-complete with respect to q, $z \leq q$, and since q is even, $\lceil z/2 \rceil \leq q/2$. Hence $wt(H,c), wt(H',c) \leq q/2$. □

Lemma 3.7 **ED** *takes time* $O(n+m)$.

proof. The first **for** loop takes time $O(m)$. From Gabow [48], **EP** takes time $O(n+m)$. The **for** loop from lines 9 - 18 also takes time $O(m)$. □

We can now state the algorithm **A2** which is based on a divide-and-conquer strategy. If q is odd, a critical k-matching of unit duration is found and deleted from the graph to make q even. If q is even, an Euler division is performed and the algorithm applied to the resulting smaller graphs. In the following, angle brackets delimit a sequence and parallel bars denote sequence concatenation.

Algorithm A2.
Input: Bipartite graph $G = (A, B, E)$ with weight function $wt : E \cup A \cup B \rightarrow N$, and integer k.
Output: A minimum-length k-coloring of G, as a sequence s of (M, b) pairs, where M is a critical k-coloring and b is its duration.

```
0. s := $\langle\ \rangle$;
1. w := max(wt(v): v $\in$ A $\cup$ B;
2. W := sum(wt(e): e ∈ E);
3. q := max(w, ⌈ W/k ⌉);
4. Add dummy traffic to make G k-complete with respect to q.
5. A2-color(G);          /* Updates s */
6. Delete dummy traffic from s.
7. end A2.
```

Procedure A2-color(G)

```
1. if q is odd then
2.     M := CKM(G);          /* CKM will find a critical k-matching */
3.     s, E, q := s || ⟨ (M, 1) ⟩, E - M, q - 1;
4. end if
5. H, H' = ED(G);
6. A2-color(H);
7. A2-color(H');
8. end A2-color
```

Theorem 3.5 *Algorithm* **A2** *finds a minimum-length schedule for an instance of DTS.*

proof. We need to show that **A2** generates a minimum-length k-coloring of G. Lines 1 - 4 and 6 - 7 of **A2** are similar to **KT**. Procedure **A2-color** is called with G, a bipartite graph k-complete with respect to its bound q, as its argument. Thus, from Theorem 3.1, G contains a critical k-matching, which can be found by the procedure **CKM** outlined in the proof of Theorem 3.1.

If q is odd, a critical k-matching is found in line 2 of **A2-color**. Now the graph G with edge set $E - M$ has an even bound $q - 1$, and from a corollary of Theorem 3.2, is also k-complete with respect to its bound. Thus from Lemma 3.6, **ED** generates an Euler division of G into H, H', which are k-complete graphs with respect to their bounds $\lfloor q/2 \rfloor$. **A2-color** is applied to them recursively, and by the induction hypothesis generates two k-colorings of length $\lfloor q/2 \rfloor$. Together with the k-matching generated if q is odd, we obtain a k-coloring in s of length q. □

Theorem 3.6 *Algorithm* **A2** *runs in time* $O(Kmn^{.5} \log n)$*, where* K *is the maximum edge weight of* G.

proof. Procedure **CKM** takes time $O(n^{.5}m)$ using the **MaxMatch** algorithm of Hopcroft and Karp [71]. By Lemma 3.7, each invocation of **ED** takes time $O(n+m)$. As long as $K > 1$, **ED** results in two subgraphs with bound $\lfloor q/2 \rfloor$, each having upto n vertices and m edges. When $K = 1$, **ED** results in two subgraphs with bound $\lfloor q/2 \rfloor$, each having roughly $m/2$ edges. Therefore, for $K > 1$, the time $T(q, m)$ for an invocation of **A2-color** with a graph of bound q and m edges is $T(q, m) = O(n^{.5}m) + 2T(\lfloor q/2 \rfloor, m) = KT(q/K, m) + (K - 1)O(n^{.5}m) \leq KT(m, m) + KO(n^{.5}m)$. Now $T(m, m) = 2T(m/2, m/2) + O(n^{.5}m)$, i.e., for some $c, m' \in \mathcal{N}$, $T(m, m) \leq 2T(m/2, m/2) + cn^{.5}m$ for all $m > m'$. Hence $T(m, m) \leq 4T(m/4, m/4) + 2cn^{.5}m/2 + cn^{.5}m \leq mT(1, 1) + (\log m)cn^{.5}m$, for all $m > m'$, i.e., $T(m, m) = O(n^{.5}m \log n)$. Therefore, the total time $T(q, m) = O(Kmn^{.5} \log n)$. □

Thus **A2** is superior to **KT** for problem classes where K is small relative to

$n^{4.5}/(m \log n)$, i.e., say smaller than n^2. **A2** is superior to **A1** when K is small relative to $n^4/(m \log n)$, say smaller than $n^{1.5}$.

3.6 An algorithm for large transfer lengths (A3)

While **A2** is satisfactory for problems where K increases at a modest rate relative to n, it does not handle problems with large transfer lengths well. In fact, **A2** is not a polynomial-time algorithm, but a pseudo-polynomial time algorithm [56]. This is because the weight of each edge in the input graph must be supplied to the algorithm.[2] In this section we describe how an algorithm for edge coloring of weighted bipartite graphs with $k = n$ [51], can be applied for DTS when $k \leq n$. The key observation is that deleting a maximal matching from a k-filled graph (see section 3.3) leaves a k-filled graph.

Consider a weighted bipartite graph $G = (A, B, E)$ with weight W, maximum vertex weight w, maximum edge weight K, and bound q. By Lemma 3.4, it can be converted to a k-complete graph with respect to q by adding dummy traffic, and then to a k-filled graph G' with respect to q using the k-**fill** algorithm This conversion is performed in **KT** and **A1**, and takes time $O(m + n)$.

[2]It takes $O(\log K)$ bits to encode the weight of each edge, so that $O(m \log K)$ bits are needed to supply the information about the edge weights in the input graph. On the other hand, it takes $O(\log n)$ bits to encode each vertex label, and hence $O((m+n)\log n)$ bits to encode the graph connectivity. Thus the total length of the string describing the input is $I = O(n^2(\log n + \log K))$. The time complexity of **A2** increases as a polynomial in I if only n increases, but as an exponential in I if K increases.

Let $G' = (A', B', E')$ with $|A| = |B| = 2n - k$, weight $W' = (2n-k)q$, maximum edge weight K', and all vertex weights equal to q. In both **KT** and **A1**, a maximal matching M' of size $2n-k$ on G' is obtained, and the k edges of $M = M' \cap E$ are deleted from G. This results in a new graph H which is k-complete with respect to $q-1$ but not k-filled with respect to $q-1$. It is necessary to invoke k-**fill** on H before obtaining the next critical k-matching. We observe that if all $2n-k$ edges of M' are deleted from G', the result is a graph G'' that is k-filled with respect to $q-1$, obviating the need for k-filling again before obtaining the next matching.

The argument above leads to the following algorithm, where an edge-coloring on G' is used to obtain a series of maximal matchings on G', which are then pruned to obtain a series of critical k-matchings on G. The **weighted-edge-coloring** algorithm of Gabow and Kariv [51] is used to edge-color G'. It should be pointed out that the space complexity of the **weighted-edge-coloring** algorithm, and hence of **A3**, is high: $O(mn \log K)$.

Algorithm A3.

Input: Bipartite graph $G = (A, B, E)$ with bound q and maximum edge weight K.

Output: A minimum-length k-coloring of G, as a sequence s of (M, b) pairs, where M is a critical k-coloring and b is its duration.

```
0. s := ⟨ ⟩;
1. Add dummy traffic to make $G$ $k$-complete with respect to $q$.
2. G' := $k$-fill(G);
```

```
3. C := weighted-edge-color(G');            /* C is an edge coloring of G'*/
4. for each color $c \in C$
5.     M', b := edges colored by c, duration c is used;    /* M' is max.
matching on G' */
6.     M := M' ∩ E;
7.     s := s || ⟨ (M, b) ⟩;
8. end for
9. Delete dummy traffic from $s$.
10. end A3.
```

Theorem 3.7 *Algorithm* **A3** *takes time* $O(n^3(\log n + \log K))$.

proof. As noted earlier, k-**fill** takes time $O(n)$. The weighted edge coloring algorithm [51] takes time $O(|V||E|\log J)$ for a weighted bipartite graph with largest edge weight J. In **A3** we apply this algorithm to G', where $|V| = 2n-k$, $|E| \leq (2n-k)^2 = O(n^2)$, and $J = K' \leq q \leq mK$. Thus the total time is $O(n^3(\log n + \log K))$ to obtain an edge-coloring of G'. Since the coloring algorithm [51] uses $O(|E|\log J)$ colors, the number of iterations of the **for** loop is $O(n^2(\log n + \log K))$. Since $|M'| = 2n - k$, lines 5 - 7 can be implemented in time $O(n)$, and thus the **for** loop takes time $O(n^3(\log n + \log K))$. □

Algorithm **A3** is faster than **KT** and **A1** for a large class of graphs, i.e., when $\log K$ is small relative to $n^{1.5}$. Note that **A2** is still faster than **A3** for $K = 1$ or when K is bounded by a small constant.

3.7 Experimental evaluation

In this section we present the results of an experimental evaluation of the algorithms described in this chapter. This work has confirmed that our algorithms provide results that are superior to the **KT** algorithm for the situations studied. This experimental work extends the previous related work of Somalwar [132], Kandappan [85], Balan [4] and Jain et al [78]. It has also led to the investigation of even faster algorithms for data transfer scheduling, which are heuristic in nature [76], and are discussed in a subsequent chapter.

We compare the effects of using the four scheduling algorithms discussed in this chapter, namely **KT**, **A1**, **A2**, and **A3**. Since all four algorithms produce optimal schedules, the key question is the amount of time taken to produce those schedules. There are four parameters that affect the performance of these algorithms: the number of vertices in each partition of the graph (n_1 and n_2), the number of edges m, the maximum number of simultaneous transfers allowed k, and the maximum edge weight K. When the number of vertices in each partition is the same, we set $n = n_1 = n_2$. We evaluate the bahavior of the algorithms as each of these parameters is varied.

Scenario: Volume Visualization of Scientific Data. Consider a scenario where users, who may be physicians, health care workers, scientists, etc., need to share and access a large image database. The images may consist of medical information, e.g. computer-aided tomography (CAT) scans, or oil prospecting information, e.g. seismic data from acoustical depth soundings, and so on. The database is processed and stored at a parallel computer site, and users view the images by requesting image files to be displayed on their graphics workstations. The parallel computer is a shared-bus system, in which processors and disks are connected to a set of common buses (or a single high-speed system bus that is shared in a time-multiplexed fashion), which allow multiple I/O transfers to proceed in parallel. In order to provide a reasonable response time for the users, the workstations are also connected to the common buses. A user request for an image file is processed by the CPUs at the parallel computer, and results in image data being transferred from the system disks to the user's workstation across the common buses. The workstations off-load some low-level image-processing tasks from the parallel computer, such as rendering, shading, etc. In this scenario, the parallel computer's operating system batches the image file requests and schedules the resulting I/O transfers.

In terms of this parallel I/O application, n_1 and n_2 correspond to the number of disks and workstations, m the number of image files to be transferred, k the number of parallel buses in the system including the degree to which they can be time-multiplexed, and K the longest file length in disk blocks. Using the parallel I/O application as a context helps to bound the ranges of values of the parameters for which the scheduling algorithms are evaluated. Given the scalability problems of shared-memory shared-bus parallel computer systems, we evaluate the algorithms for relatively modest numbers of disks and workstations ($n_1, n_2 < 256$). Typically, we choose $n = n_1 = n_2 = 64$. Consistent with this context, we also assume only a relatively modest number of simultaneous parallel transfers ($4 \leq k \leq 16$, and $k < n_1, n_2$). As far as the number of transfers is concerned, we choose $100 \leq m \leq 1000$ as a reasonable range considering the image database scenario sketched above. The maximum file length, K, is increased systematically until the behavior of the program seems to become clear from the trend of the increase in CPU time.

The algorithms were implemented as programs in C, generally following the outlines sketched in this chapter. The implementation of **KT** and **A2** was a non-trivial adaptation from the implementation of Somalwar [132], which handles unit-weight edges only. The Hopcroft and Karp [71] maximum cardinality matching algorithm used in **A1** and **A2** was implemented in a form very similar to that given in the text by Moret and Shapiro [106]. In the implementation of **A2** we made two modifications: the recursive structure of the program was replaced by iteration, and rather than performing k-filling at every iteration, it was performed only at the start of the algorithm.

The main implementation difficulty was with the **weighted-edge-color** algorithm of Gabow and Kariv [51], especially since their description omits two important points. The first is regarding the conditions under which new colors are assigned to edges; it was necessary to carry out a detailed case analysis of the situations in which the weighted augmenting path algorithm should in fact assign new colors to uncolored edges, taking into account the special cases that occur when all the remaining uncolored edges are of unit weight. The second omission was more serious, and was the key observation that as one weighted colored edge is partially assigned a new color, all edges in the graph bearing the old color must also be partially assigned the new color. Again, special cases arise when all uncolored edges are unit-weight. These two points are very important as they form the basis for guaranteeing that the number of colors used in **weighted-edge-color**, and hence its time complexity, is logarithmic in K rather than linear. Implementing **weighted-edge-color** to correctly handle these two points is quite involved.

The programs implementing **KT** and **A1-A3** were evaluated by measuring the CPU time they take to execute when presented with uniformly randomly generated bipartite graphs as inputs. Random graphs were generated for selected combinations of the n_1, n_2, m and K program parameters using a pseudo-random number generator [94]. The programs were executed on a Sun Sparc 2 workstation running the SunOS^{TM} Release 4.1.1 operating system, after being compiled using the Sun Microsystems C compiler (bundled with SunOS Release 4.1.1), with Level 4 optimization enabled ("-O4" option). The data structures for the programs all fit in the 32 MB main memory of the workstation, and so the programs do not perform any I/O in order to execute, except to read the input graph and print results.

The CPU time taken by a program for each input random graph was measured by the C Shell "time" command. Although this measurement tool has a resolution of only 20 ms, it was not thought necessary to use a higher resolution measurment (e.g. the system's real-time clock) since most measurements we take are on the order of seconds and the programs tend to display rather pronounced differences in their execution times. For each selected combination of the program parameters, one hundred random graphs were generated, and the CPU time taken by each program, as reported by the "time" command, was recorded for each input graph. The mean and standard deviation of each set of 100 measurements was calculated using the programs given in *Numerical Recipes* [119]. The data was plotted using the DeltaGraph Professional^{TM} software package, on a Macintosh system. The same software was also used to generate smooth curve fits based on models that were supplied to the program as candidates.

In the following we present the results of the experiments, the calculated means, and plotted points and curve fits for each progream as each of the four parameters, m, m, n and K were varied. Although the data presented here are for $n = n_1 = n_2$, in a previous study we have considered the case $n_1 \neq n_2$ and found qualitatively similar results [78].

3.7.1 Effect of varying the number of transfers

In Fig. 3.1 we plot the mean CPU time taken by each program implementing **KT**, **A1**, **A2** and **A3** for inputs where the parameters $n = n_1 = n_2 = 64$, $k = 4$ and $K = 1$ are fixed and m varies from 100 to 1000. That is, we see the effect of varying the number of transfers while keeping all other parameters fixed. Each data point represents the mean of 100 measurements of CPU time, and the error bars indicate one standard deviation above and below the mean. The curve-fits shown in Fig. 3.1 correspond to the following equations and correlation coefficients:

$$KT(t) = 1.55 \times 10^{-6}\, m^2 + 2.18 \times 10^{-3}\, m - 2.18 \times 10^{-2}\ ,$$

$$R2^2 = .98,\ R1^2 = .99,\ R0^2 = .99$$

$$A1(t) = 1.73 \times 10^{-6}\, m^2 + 1.48 \times 10^{-3}\, m - 6.66 \times 10^{-2}\ ,$$

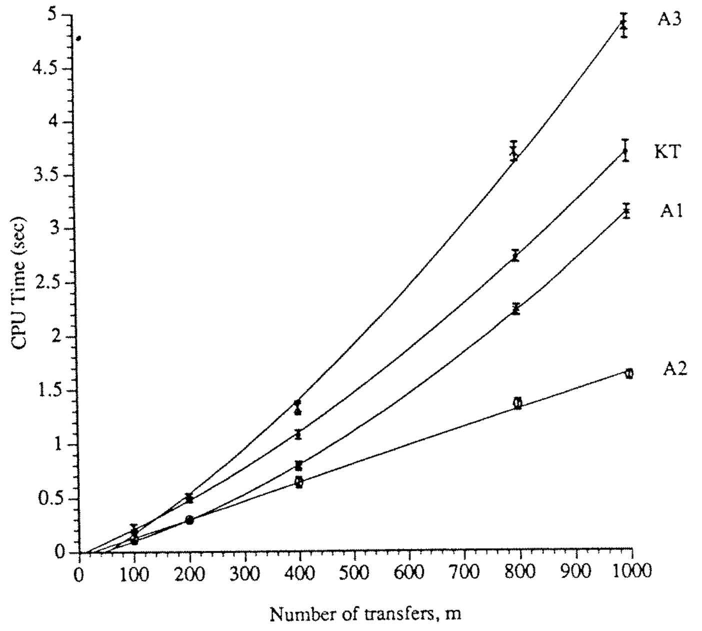

Figure 3.1: CPU time versus number of transfers for $n = 64$, $k = 4$, $K = 1$

$$R2^2 = .99, \; R1^2 = .98, \; R0^2 = .99$$

$$A2(t) = 1.69 \times 10^{-3}\, m - 3.82 \times 10^{-2}\,, \qquad R^2 = .99$$

$$A3(t) = 7.37 \times 10^{-4}\, m \log m - 0.258, \qquad R^2 = .99$$

The results plotted and curve-fitted in Fig. 3.1 display some interesting characteristics. It is interesting to see that the execution time of **A2** in these experiments increases very close to linearly with m, exactly as predicted by the theoretical worst-case time complexity formula $O(Kmn^{.5}\log n)$ derived for **A2** in the previous section. On the other hand, we see that the other three algorithms also show marked increases with m, which are not predicted by the theoretical complexity analysis. This discrepancy is discussed in the next section of this chapter.

Qualitatively, however, we see that for this combination of parameters, **A2** out-performs the other algorithms significantly, and is likely to continue doing so as m increases.

3.7.2 Effect of varying the degree of data transfer parallelim

In Fig. 3.2 the parameters $n = n_1 = n_2 = 64$, $m = 1000$, and $K = 1$ are fixed and k varies. That is, we see the effect of varying the degree of parallelism in the data transfer while keeping all other parameters fixed. The equations corresponding to the curve fits are given by:

$$KT(t) = 15.33\ k^{-1.02}, \qquad R^2 = .99$$

$$A1(t) = 12.23\ k^{-0.98}, \qquad R^2 = .99$$

$$A2(t) = 5.13\ k^{-0.84}, \qquad R^2 = .99$$

$$A3(t) = 30.49\ k^{-1.33}, \qquad R^2 = .99$$

For all four algorithms, the CPU time varies inversely with k, a trend not predicted by the theoretical time complexity formulas derived earlier. It is interesting to see that for all practical purposes the variation is proportional to $1/k$ for **KT** and **A1**. This is discussed in the next section.

Qualitatively, we again observe that for this set of parameters, **A2** out-performs the other algorithms signficantly.

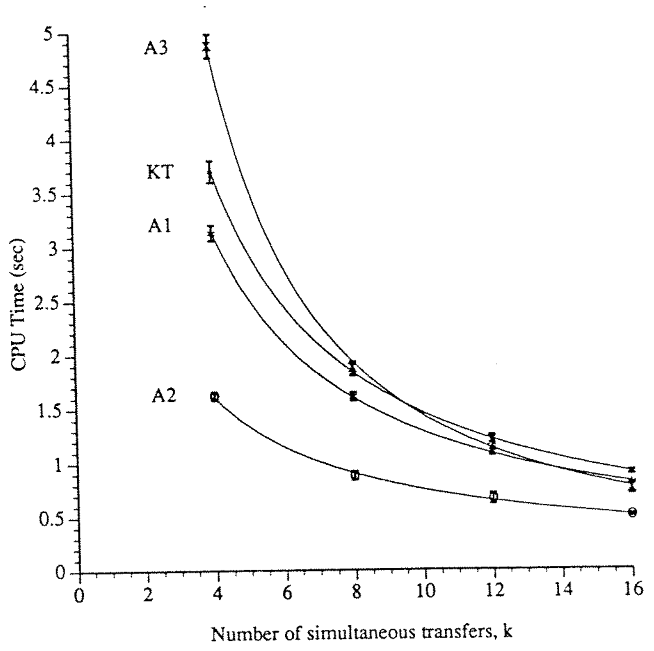

Figure 3.2: CPU time versus number of simultaneous transfers for $n = .64$, $m = 1000$, $K = 1$

3.7.3 Effect of varying the number of resources

In Fig. 3.3 the parameters $k = 4$, $m = 1000$, and $K = 1$ are fixed and $n = n_1 = n_2$ is varied. That is, we see the effect of varying the number of resources in the system while keeping all other parameters fixed. The curve fits are given by:

$$KT(t) = 4.45 \times 10^{-4}\, n^2 + 1.01 \times 10^{-2}\, n + 1.26,$$

$$R2^2 = .99,\ R1^2 = .96,\ R0^2 = .99$$

$$A1(t) = 1.17 \times 10^{-3}\, n^{1.5} + 1.91 \times 10^{-1}\, n^{0.5} + 1.26, \qquad R^2 = .99$$

$$A2(t) = 1.18 \times 10^{-2}\, n \log n + 1.55, \qquad R^2 = .99$$

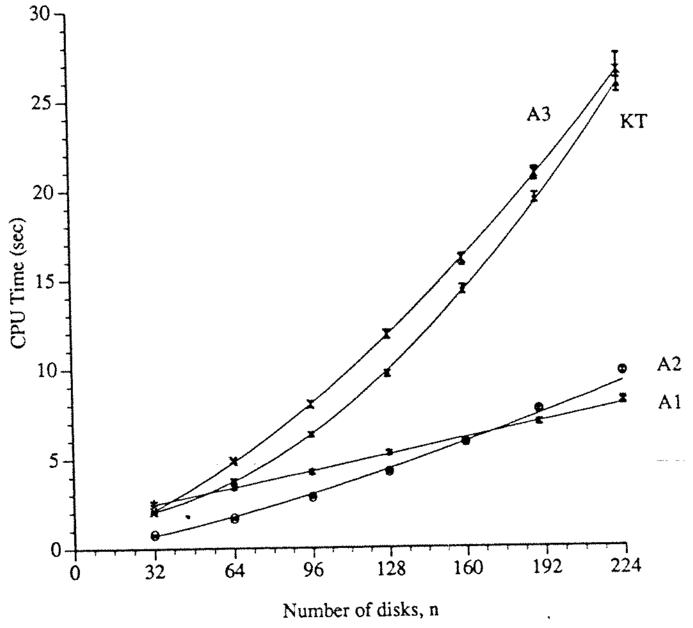

Figure 3.3: CPU time versus number of resources for k = *4, m = 1000, K = 1*

$$A3(t) = 2.75 \times 10^{-4}\ n^2 + 5.68 \times 10^{-2}\ n + 0.02,$$

$$R2^2 = .99,\ R1^2 = .98,\ R0^2 = .99$$

We observe that, at least for the set of experiments described here, the performance of both **KT** and **A1** appears to be significantly better than that predicted by their theoretical complexity formulas $O(n^5)$ and $O(n^{4.5})$ respectively. This is discussed in the next section.

Qualitatively, we observe that for this set of parameters **A1** and **A2** significantly out-perform the other algorithms, with **A1** likely to have better performance than **A2** only for $n > 160$.

3.7.4 Effect of large transfer lengths

In Fig. 3.4 the parameters $n = 64$, $k = 4$, and $m = 1000$, are fixed while K is varied. Thus the same number of transfers have to take place for all runs, but their lengths are integers drawn uniformly at random from the interval [1, K]. The curve fits are given by:

$$KT(t) = 0.23 \log K + 0.25. \qquad R^2 = .96$$

$$A1(t) = 0.14 \log K + 0.13 \qquad R^2 = .95$$

$$A2(t) = 0.08K + 0.09, \qquad R^2 = .99$$

$$A3(t) = -8.65 \times 10^{-4}\ K^2 + 0.18K + 0.29,$$

$$R2^2 = .82,\ R1^2 = .96,\ R0^2 = .99$$

We observe a number of interesting features in this set of curves. The first is that although the CPU time behavior of both **KT** and **A1** can be fit to an $O(\log K)$ curve, the constants involved are so small that it is essentially independent of K for $K > 10$. This is as predicted by the theoretical time complexity analysis. We also see that the CPU time for **A2** increases very close to linearly with K, again as predicted by theoretical analysis. On the other hand, the behavior of **A3** does not follow $O(\log K)$ for this range. This is discussed in the following section.

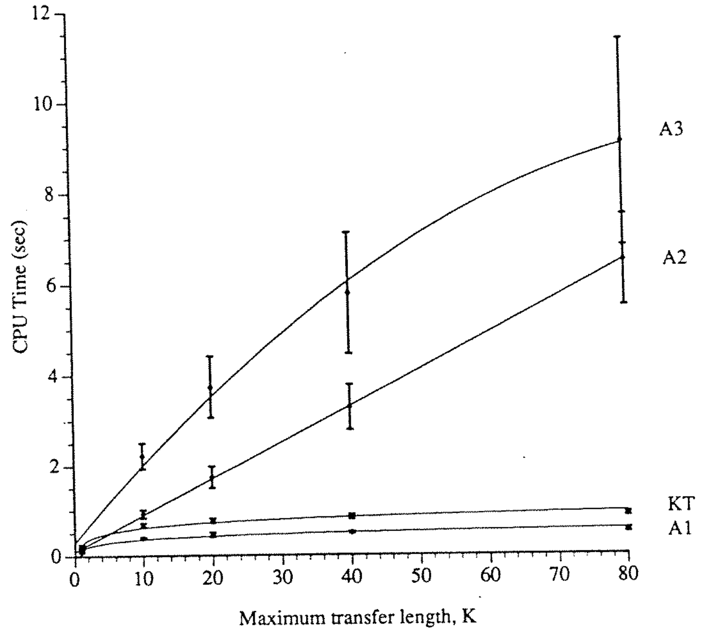

Figure 3.4: CPU time versus maximum transfer length $n = 64$, $k = 4$, $m = 1000$

Qualitatively, we observe that **KT** and **A1** significantly out-perform the other algorithms, both in terms of absolute CPU time and in terms of its variance for random input instances. Between the better two algorithms, **A1** consistently out-performs **KT**.

3.8 Interaction of theoretical and experimental evaluation

Our study of the four scheduling algorithms **KT**, and **A1** - **A3**, is an interesting example of the importance of cross-checking theoretical and experimental evaluations of algorithm behavior. In several sets of experiments described above, we found that the measured time behavior of the algorithm differs significantly from that predicted by theoretical analysis alone. These discrep-

Algorithm	Method	m	k	n	K
KT	theory	const.	const.	n^5	const.
	experiment	m^2	$1/k$	n^2	const.
A1	theory	const.	const.	$n^{4.5}$	const.
	experiment	m^2	$1/k$	$n^{.5} + n^{1.5}$	const.
A2	theory	m	const.	$n^{.5} \log n$	K
	experiment	m	$1/k^{.84}$	$n^{.5} \log n$	K
A3	theory	const.	const.	$n^3 \log n$	$\log K$
	experiment	$m \log m$	$1/k^{1.3}$	n^2	K^2

Table 3.1: Asymptotic theoretical vs. experimental behavior of algorithms as input parameters vary. (See following Table also)

ncies motivate us to re-examine our theoretical and experimental evaluation more closely, leading to a better understanding of the algorithms' behavior.

The discrepancies we observe are summarized in Table 3.1, where we show the asymptotic theoretical and experimental behavior of the algorithms as each of the parameters m, k, n and K is varied. (The curve-fitted constants have been omitted from the experimental results as we are only considering the estimated asymptotic algorithm behavior). In this section we show that most of these apparent discrepancies can in fact be resolved by one of two ways: more sophisticated theoretical analysis, and further experimentation.

3.8.1 Effect of number of transfers

We first consider the marked discrepancy between theoretical and experimental behavior of **KT** and **A1** as m is varied. We re-examine the theoretical time complexity analysis presented earlier in this chapter and observe that it can be made more precise in two ways.

Observation. The time complexity of the max-min bipartite weighted matching algorithm [92], **MaxMinMatch** can be refined to $O(mn)$ from $O(n^3)$. Similarly, the time complexity of the maximum cardinality matching algorithm **MaxMatch** [71] can be refined to $O(mn^{.5})$ from $O(n^{2.5})$. $\square$

Lemma 3.8 *The number of times that critical k-matchings have to be calculated in either* **KT** *or* **A1***, i.e., the number of iterations L, can be refined to $L = O(m+n)$ from $L = O(n^2)$.*

proof. Recall that a new critical k-matching must be calculated, in both **KT** or **A1**, if either the weight of one of the edges covered by the matching decreases to zero, or a vertex that was previously not critical weight becomes critical. Thus at every iteration of the algorithm, either an edge can be deleted, or a vertex becomes critical, or both. Now observe that once a vertex becomes critical it remains critical until the algorithm terminates (since it must continue to receive full service if the algorithm is to produce a schedule which meets the lower bound on schedule length). Hence any vertex can be promoted from being non-critical to critical at most once during the execution of the algorithm. Also, any edge can be deleted at most once. Since at each iteration at least one of these events (edge deletion or vertex promotion) occurs, there are at most $O(m+n)$ iterations. $\square$

A careful reading of the proof in [9] shows that the authors have used a similar reasoning to the one we have presented above, but have set $m = O(n^2)$, leading to the $L = O(n^2)$ bound.

Theorem 3.8 *The time complexity of algorithm* **KT** *can be refined to* $O(m^2n + mn^2)$ *from* $O(n^5)$*, and of algorithm* **A1** *to* $O(m^2n^{.5} + mn^{1.5})$.

proof. The time complexity of both **KT** and **A1** is given by L times C, the time for the critical k-matching algorithm used. From the observation, $C = O(mn)$ for **KT** and $C = O(mn^{.5})$ for **A1**. From Lemma 3.8, $L = O(m + n)$. The theorem follows. □

We can use this result to explain the experimentally observed variation of **KT** and **A1**'s running time not only with m, but with n.

We now consider the discrepancy between the theoretical time complexity of **A3**, $O(n^3(\log n + \log K))$, and the experimentally observed variation with m. Recall that **A3** performs k-**fill** on the input graph G with n vertices and m edges to produce a graph G' which is then colored using the **weighted-ege-color** algorithm of Gabow and Kariv [51]. We first show that k-filling G only increases the number of edges in G' to $O(m + n)$ instead of $O(n^2)$.

Lemma 3.9 *A bipartite graph with n vertices and m edges can be converted to a k-filled graph with at most $O(m+n)$ edges, assuming that weighted edges can be used for k-filling.*

proof. Let $G = (A, B, E)$ be the input graph, w its vertex weight and W its total weight. For ease of exposition we assume $|A| = |B| = n$ in the following; the case $|A| \neq |B|$ is very similar and left to the reader.

From the time complexity analysis of the k-**complete** algorithm we see that at most $O(n)$ edges are added during this phase. The k-filled bipartite graph $G' = (A \cup C, B \cup D, E \cup E')$ is obtained by adding $n - k$ vertices to each partition of G and by adding edges $E' \subseteq (A \times D) \cup (B \times C)$ until all vertices have weight q.

It has been shown that k-filling can always be done [10]. We show by induction the proposition that k-filling G requires adding at most $4n$ edges, as follows. It suffices to consider vertices in only one partition of G, say A, with the number of its vertices examined, j, being the induction variable. Initially the first vertex $a_1 \in A$ has been examined. An edge of weight $q - wt(a_1)$ is added from a_1 to d_1, the first vertex in D; the proposition holds. Assume that after j vertices have been examined, at most $2j$ edges have been added. When a_{j+1} is examined, let d_k be the first vertex in D with weight less than q. An edge (a_{j+1}, d_k) of weight $w' = q - \max(wt(a_{j+1}, wt(d_k))$, and another edge (a_{j+1}, d_{k+1}) of weight $q - w'$, are all that are needed to make $wt(a_{j+1}) = q$. Thus the induction is complete and the proposition holds.

It follows that if non-unit-weight edges are used for k-filling, G' has $O(m+n)$ edges. □

Notice that making a graph k-filled with respect to its bound increases not only its number of edges but also the maximum edge weight. In fact, during k-**complete**, for instance, the maximum edge weight can increase from K to as much as mK. To see an example of this, consider the graph with $n = 5$, $m = 4$ edges each of weight $K = 2$ connected from a single vertex $a \in A$

to four distinct vertices in B, and given $k = 3$. Then $w = W = 8$, and $q = 8$, so that edges of total weight $kq - W = 16$ need to be added while not increasing the maximum vertex weight. This can be done by adding three edges, of weights 8, 6 and 2, respectively, making $K' = 8 = mK$. However, the the bound of the graph, q, remains unchanged, and so the time complexity analysis of **A3** and other algorithms is not affected.

Theorem 3.9 *The time complexity of* **A3** *can be refined to* $O((nm + n^2)$ $(\log m + \log K))$ *from* $O(n^3$ $(\log n + \log K))$.

proof. The **weighted-edge-color** algorithm of [51] takes time $O(|V'|\ |E'|$ $\log J)$ where V' is the total number of vertices and J is the maximum edge weight of G'. While $|V'| = O(n)$ and $J \leq mK$, from Lemma 3.9 we have $|E'| = O(m + n)$, giving the new complexity estimate. A similar argument holds for the **while** loop in the **A3** algorithm. □

As an aside, we make the following observation here, which will be used in a later section.

Observation. If only unit-weight edges can be used for k-filling, G' can have upto $O(nm)$ edges. □

3.8.2 Effect of data transfer parallelism

We consider the effect of increasing k, the number of simultaneous transfers posssible, upon the behavior of the four scheduling algorithms.

Informally, we can consider this effect by recalling that the length of the schedule is given by the bound of the graph, $q = \max(w, \lceil W/k \rceil)$. In general, as k decreases the second component of q dominates, and the schedule length increases. Since all four algorithms are essentially iterative computations of matchings (or, in the case of **A3**, of augmenting paths), and the number of iterations increases with the schedule length, as k decreases the execution time of all four algorithms increases. For situations where $\lceil W/k \rceil > w$, the CPU time for all four algorithms should vary as $1/k$.

3.8.3 Effect of transfer lengths

The theoretical advantage of **A3** over **A2** is that its time complexity is polynomial in K rather than a pseudo-polynomial. However, while **A2**'s CPU time increases linearly with K as expected for $1 \leq K \leq 80$, **A3** appears to perform much worse.

We observe from the variation of **A3** with m, n, and k that it appears to have much larger constants of variation than the other three algorithms. We thus extended the investigation of **A3**'s behavior to larger values of K in order to estimate its asymptotic behavior. However, as noted earlier, **A3**'s storage requirements are very high. In order to keep all data structures in memory, experiments for $K > 80$ could not be conducted on the Sun Sparc 2 workstation, which has 32 MB of main memory. The program was run instead on a Solbourne Series5e/900$^{\text{TM}}$ workstation with 128 MB of main memory. The workstation runs Solbourne's UNIX-like OS/MP 4.1A.1 and can run programs compiled for the Sun Sparc 2 without recompilation. Input

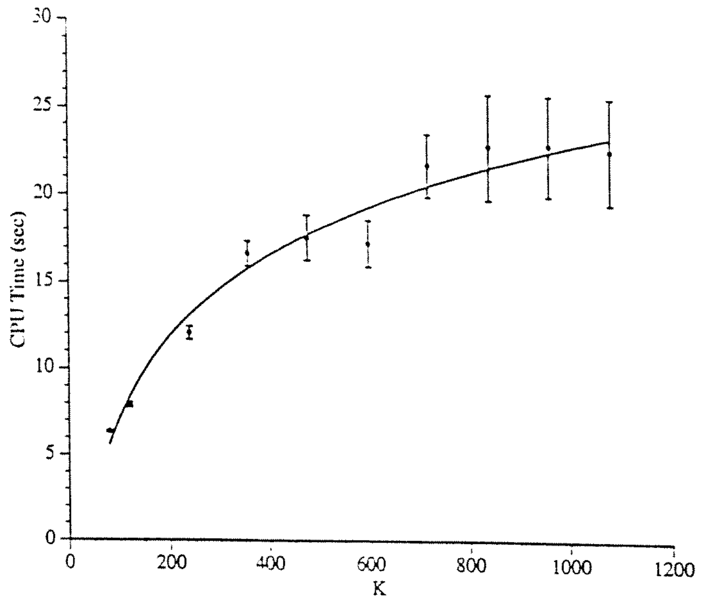

Figure 3.5: CPU time on Solbourne versus maximum transfer length for $n = 64$, $k = 8$, $m = 100$

graphs, time measurements, and mean CPU times were generated as before, and plotted with error bars and curve fits as before, to yield the plot in Fig. 3.5. Notice the relatively large standard deviation of the measurments. The curve fit for $K \geq 80$ is given by:

$$A3(K) = 9.83 \log K - 24.2, \qquad R^2 = 0.96$$

The revised comparison of theoretical and experimental results in Table 3.2 displays good agreement between the two.

Algorithm	Method	m	k	n	K
KT	theory	m^2	$1/k$	n^2	const.
	experiment	m^2	$1/k$	n^2	const.
A1	theory	m^2	$1/k$	$n^{.5} + n^{1.5}$	const.
	experiment	m^2	$1/k$	$n^{.5} + n^{1.5}$	const.
A2	theory	m	$1/k$	$n^{.5} \log n$	K
	experiment	m	$1/k^{.84}$	$n \log n$	K
A3	theory	$m \log m$	$1/k$	n^2	$\log K$
	experiment	$m \log m$	$1/k^{1.3}$	n^2	$\log K$

Table 3.2: Revised asymptotic theoretical vs. experimental behavior of algorithms as input parameters vary

3.9 Discussion

3.9.1 Previous related work

Algorithms **A1 - A3** provide a method for scheduling parallel data transfers such as I/O in multiple-bus parallel computers and time slots in TDMA switches, and are a substantial improvement in both theoretical and experimentally observed execution time over algorithm **KT**. Since **A1 - A3** are framed in terms of optimal edge-coloring of bipartite multigraphs, we review this literature briefly. In Table 3.3 we summarize the previous work. We emphasize that almost all the literature surveyed consists of theoretical work only. Very few studies of the type reported in this chapter have been performed to experimentally evaluate the behavior of the scheduling and coloring algorithms that have been developed.

First consider the optimal edge-coloring of bipartite graphs with unit-weight edges. Vizing [137] developed an algorithm using the basic notion of augmenting paths that takes time $O(mn)$. Gabow [48] exploited the partitioning of a bipartite graph by means of Euler partitions in order to apply the divide-and-conquer strategy to edge coloring algorithms. This idea has been used repeatedly [50, 51, 29, 132, 78]. In the same paper Gabow noted that the Mendelsohn-Dulmage method (see [92]) could be used to construct an algorithm to obtain a matching covering all vertices of maximum degree. This matching algorithm takes time $O((m+n)n^{0.5})$ and was used in conjunction with the divide-and-conquer strategy to obtain an edge coloring algorithm of time complexity $O(n + mn^{0.5}\log n)$.

Gabow [48] also observed that in the special case that the degree of the graph is a power of 2, the divide-and-conquer algorithm need never call the matching algorithm, allowing an edge coloring to be found in time $O(n + m\log n)$. Gabow and Kariv [50, 51] use this observation to obtain faster edge-coloring algorithms for graphs whose degree is not a power of 2, by repeatedly constructing partially-colored subgraphs whose degree is a power of 2 and coloring them efficiently.

Finally, Cole and Hopcroft [29] use the idea of Euler partitioning the graph in order to design a matching algorithm that covers all vertices of maximum degree. This matching algorithm runs in time $O(\max(m, n\log n\log^2 d))$, where d is the graph degree, and is faster than the Mendelsohn-Dulmage matching method used by Gabow [48] by a factor of roughly $O(n^{0.5})$; it leads to an $O(m\log n)$ algorithm for edge coloring.

There has been relatively little attention paid to the problem of edge-coloring weighted bipartite graphs. Before we review this literature, it is interesting

Reference	Unit-Weight Edges *(K = 1)*	Unequal Weight Edges *(K ≥ 1)*
Unlimited transfers, $k = n$:		
Vizing, 1964 [137]	mn	
Gonzalez and Sahni, 1976 [62]	m^2	m^2
Gabow, 1976 [48]	$n + mn^{0.5} \log n$	
Gabow and Kariv, 1978 [50]	$mn^{0.5} \log n$	
Gabow and Kariv, 1982 [51]	$m \log^2 n$; $n^2 \log n$	$nm \log K$
Cole and Hopcroft, 1982 [29]	$m \log n$	
Limited transfers, $k \leq n$:		
Bongiovanni et al, 1981 (**KT**) [9, 10]	$m^2 n + mn^2$	$m^2 n + mn^2$
Somalwar, 1988 [132]	$mn^{1.5} \log n$	
A1	$m^2 n^{0.5} + mn^{1.5}$	$m^2 n^{0.5} + mn^{1.5}$
A2	$mn^{0.5} \log n$	$Kmn^{0.5} \log n$
A3	$(n^2 + nm) \log m$	$(n^2 + nm)$ $(\log m + \log K)$

Table 3.3: Summary of previous related work

to discuss an influential early application, that of constructing class-teacher timetables [63], also called the timetabling problem [8, 31]. To quote Gotlieb in 1962, "For a high school with thirty or more classes, even after the sets of teachers to be associated with a given class are assigned, it takes many man-weeks to draw up a schedule specifying when teachers and classes are to meet. ... [I]n the Metropolitan Toronto area alone, over sixty large time-tables are drawn up annually" [63].

The class-teacher timetabling application in its simplest form can be modeled as a problem of edge-coloring a weighted bipartite graph, where the vertex partitions represent teachers and classes, the edges represent the teachers assigned to a class, and edge weights represent the number of contact hours [8]. The application as specified by Gotlieb included the additional practical constraints that the total number of time slots available is fixed and that for certain time slots either a class or a teacher is unavailable; for these constratints the scheduling problem is NP-complete [40]. However, Gotlieb's application has continued to attract attention because of its practical importance and theoretical applicability [31, 142].

It is interesting to note that possibly the earliest efficient algorithm for edge-coloring weighted bipartite graphs [62] was developed in the context of a scheduling application, namely the scheduling of jobs in an open shop[3] Gonzalez and Sahni's algorithm [62] finds shortest augmenting paths to obtain matchings, a strategy similar to that used in Hopcroft and Karp's matching algorithm [71], and thereby obtain an edge coloring algorithm that runs in time $O(m^2)$. Gabow and Kariv [51] again exploit the observation that graphs whose degree is a power of 2 can be colored efficiently to obtain an algorithm that takes time $O(nm \log K)$, where K is the largest edge weight; this is faster than the algorithm of Gonzalez and Sahni [62] for a large class of graphs, i.e., whenever $n \log K = o(m)$.

The problems we address in this chapter, *SimpleDTS* and *DTS*, are modeled as optimal k-coloring of a bipartite graph, i.e., optimal edge coloring a bipartite graph where each color can be used to color at most k edges. Optimal k-coloring also directly models the class-teacher timetabling application if at most k classrooms are available at any given time [8]. However, to our knowledge, the only previous algorithmic solutions for optimal k-coloring of

[3]An open shop is the job-shop problem (defined in Chapter 2) with the restriction that the precedence graph has no edges, i.e., tasks can be performed in any order.

bipartite graphs are by Bongiovanni et al [10] and Somalwar [132]. Algorithm **KT** of Bongiovanni et al [10] can operate on graphs with non-unit edge weights, and, as described in this chapter, takes time $O(m^2n + mn^2)$. The algorithm of Somalwar [132] is applicable only to graphs with unit edge weights, and takes time $O(n^{1.5}q \log q)$, where q is the bound of the graph, i.e., takes time $O(mn^{1.5} \log n)$. As shown in this chapter, our algorithms **A1** - **A3** are faster than either of these algorithms.

It is interesting to compare the performance of the algorithm of Somalwar [132] and **A2** for unit-weight edges (see Table 3.3), since **A2** basically extends Somalwar's algorithm to the unequal weight edges case. The reason that **A2** is faster by a factor of n, even if only unit-weight edges are allowed in the input graph, is that **A2** can handle weighted edges being introduced by k-**fill**, while Somalwar's algorithm cannot. Thus for **A2** the k-filled graph has only $O(m+n)$ edges instead of $O(mn)$ edges (see Lemma 3.9 and the Observation following it). The increase in the number of edges that must be processed by Somalwar's algorithm accounts for its time complexity being higher by $O(n)$.

It is for this reason also that any simplistic application of the k-filling technique to a previous unit-weight edge coloring algorithm will result in an algorithm that performs worse than **A2**. For example, consider a simple extension of Cole and Hopcroft's [29] algorithm to handle $K > 1$ and $k \leq n$. Firstly, weighted edges will be replaced by unit-weight edges, making the number of edges $O(mK)$. Then the k-filling technique will increase the number of edges to $O(mnK)$, yielding an $O(Kmn \log n)$ algorithm for edge coloring, which is worse that **A2** by a factor of $O(n^{0.5})$. (As an aside, we observe that Cole and Hopcroft's algorithm can be extended in this way to perform better than Somalwar's).

Similarly, a simple extension of Gonzalez and Sahni's [62] algorithm to handle $k \leq n$ by k-filling will result in an algorithm that is slower than **A2** for bipartite graphs with unit-weight edges by a factor upto $O(n)$; for weighted edges it will be faster than **A3** by a factor $O(\log n + \log K)$ for sparse graphs, and slower by a factor $O(n/(\log n + \log K))$ for dense graphs. (However, such an extension to Gonzalez and Sahni's algorithm may be useful in practice, as it is relatively simple to implement and takes much less space than **A3**).

In summary, algorithms **A1** - **A3** generalize previous edge coloring algorithms [137, 62, 48, 50, 51, 29, 132] by allowing non-unit edge weights as well as a restriction on the number of edges that may be colored with a single color. The only algorithm that is as general as **A1** - **A3** is **KT** [9, 10], and as shown in this chapter, our algorithms out-perform it both in terms of theoretical and experimentally-measured performance.

3.9.2 Conclusions and future work

A key question that arises at this point is: which algorithm should be used for data transfer scheduling in bipartite architecture graphs, and under which operating conditions? Our theoretical and experimental results show that for all the situations considered in this chapter, either **A1** or **A2** should be used over the previous best algorithm, **KT**. In general, **A2** is the algorithm of choice unless either the maximum transfer length K is greater than a small constant, or the number of communicating entities (disks, transmitters, etc) n is very high, in which cases **A1** should be used.

It is interesting to consider the poor observed performance of **A3**. This seems to be because of two reasons. Firstly, the conditions under which its experimentally measured performance was better than **A2** were very limited: a relatively small number of transfers m, of large lengths K, to be carried out between relatively few entities n, with a high degree of parallelism k. This is because the constants in **A3**'s asymptotic time complexity seem to be very high. Secondly, the theoretical worst-case space complexity of **weighted-edge-color**, and hence **A3** is very large: $O(mn \log K)$ [51]. Our experiments show that in fact even on average the amount of space required is unacceptably high for most situations of practical interest. For instance, 32 MB of main memory were not sufficient to handle input graphs with parameters greater than $n = 64, k = 4, m = 1000$, and $K = 80$. Performance considerations aside, it was found that **A3** was more difficult and time-consuming to implement than any of the other algorithms, and consisted of about twice as many lines of C code. We conclude that although the algorithmic technique underlying **weighted-edge-color** is elegant and of theoretical interest, it does not lead to a practical algorithm for our application.

For future work, there are two interesting questions. The first is whether Gonzalez and Sahni's algorithm [62] can indeed be extended to solve DTS, either by using the k-filling technique or by some other means, and if so, whether its performance when implemented is fast enough to make it an attractive practical solution to DTS for interesting applications. The second is whether k-filling is needed at all for optimal k-coloring of bipartite graphs, i.e., perhaps this constraint can be satisfied at a lower level in the coloring algorithm, say at the level of finding augmenting paths.

To summarize our contributions in this chapter, we have developed and experimentally evaluated three new algorithms for scheduling data transfers in communications and parallel computer systems. These algorithms apply to a significant class of applications, such as satellite data transfers and parallel I/O due to 3D visualization software. The algorithms apply to a common class of architectures, including satellite TDMA switches, and shared-bus multiprocessors such as the Sequent [100], Encore Multimax [143] and the IBM RP3 [115]. Our theoretical and experimental investigations show that our algorithms perform significantly better than the previous best available algorithm, **KT**. Our algorithms also generalize previous theoretical work on edge-coloring algorithms for bipartite graphs [137, 62, 48, 50, 51, 29, 132], both by allowing weighted edges and restrictions on the number of edges that may be colored with a single color. Finally, to our knowledge, ours is the only extensive experimental study of the behavior of four edge-coloring algorithms for bipartite graphs in which the effects of varying the problem parameters are investigated systematically. Such experimental studies are very valuable from a practical standpoint; for example, we have shown that the practical usefulness of the **weighted-edge-color** algorithm [51] used in **A3** is severely limited both in terms of space and time cost.

Chapter 4

Heuristics for Scheduling in Bus Architectures and TDM Switches

In the previous chapter we discussed optimal algorithms for scheduling data transfers in shared-bus multiprocessors and single TDMA switches. While the algorithms we developed, **A1 - A3**, are in general faster than previous optimal algorithms like **KT** [10], we would like to have even faster algorithms, since in most applications scheduling algorithms are executed repeatedly, and any gain in speed helps overall system performance.

In this chapter we turn our attention to approximation algorithms (or heuristics) for the *SimpleDTS* and *DTS* problems restricted to unit-length transfers. Two simple greedy heuristics for unit-length transfers, **HDF** and **HCDF**, have been proposed and experimentally evaluated by Somalwar [132]. In graph-theoretic terms, these heuristics are essentially approximation algorithms for edge-coloring bipartite graphs with unit-weight edges. Both heursitics performed well in experiments, both with random input graphs as well as input graphs simulating the projected parallel I/O workload generated by applications such as 3D visualization and split-step migration. For instance, for experiments using random bipartite graphs as inputs, one of the heuristics always generated the exact solution, while running in less than 10% of the time taken by an optimal algorithm [132].

While Somalwar's result is encouraging, it is obvious that experimental evaluation can only examine a small number of combinations of the input parameters, and thus explore only a tiny fraction of the input space. In this chapter we present the first analysis of the worst-case execution time of the heuristics, as well as an analysis of their divergence from the optimal solution in the worst case. We will quantify the divergence from the optimal solution by finding a performance guarantee for each algorithm, defined as follows.

Def. If an approximation algorithm produces a schedule of length $L'(RG)$ for a problem instance with resource graph RG, and $L(RG)$ is the optimum schedule length, then the *performance guarantee* of that algorithm is $P(n)$, where $P(n)$ is the maximum value of the ratio $L'(RG)/L(RG)$, over all RG with at most n vertices.

(For the algorithms in this chapter, we will be actually be interested in $P(d)$, where d is the degree of RG.) In sec 4.1 we derive a bound for the worst-case time complexity and the performance guarantee for Somalwar's heuristics [132]. We remark that this bound is tight, i.e., it is possible to systematically generate graphs for which the heuristic performs as badly as the worst-case. In section 4.3 we compare Somalwar's experimental results [132] with the theoretical results, and finally we end with a discussion.

4.1 The Highest Degree First (HDF) Heuristic

We define the $Unit - SimpleDTS$ problem to be the $SimpleDTS$ problem restricted to the case where all tasks have unit length, i.e., for all $t \in T$, $Lp(t) = 1$. By the observations made in the previous chapter, in graph-theoretic terms, $Unit - SimpleDTS$ corresponds to the problem of finding a minimum edge-coloring of a bipartite graph with unit-weight edges. Similarly, $Unit - DTS$ corresponds to finding a minimum edge-coloring for a bipartite graph with unit-weight edges given that at most $k \leq n$ edges may be colored with a single color. We analyze the behavior of **HDF** first for $Unit - SimpleDTS$, and then for $Unit - DTS$.

4.1.1 The $Unit - SimpleDTS$ Problem

We introduce some additional terminology. Vertices $a, b \in A \cup B$ of a graph $G = (A, B, E)$ are called *partners* if $(a, b) \in E$. A vertex is said to be *colored with color c* if some edge incident upon it is colored c. Note that an edge has a unique color but a vertex may have multiple colors. A vertex is said to be *fully colored* if every edge incident upon it is colored. The degree of a vertex v is denoted $d(v)$. The degree of the graph G by $d(G)$, or simply d if clear from context. A sequence is denoted by angle brackets.

Somalwar's Highest-Degree-First **HDF** heuristic for $Unit - SimpleDTS$ for a graph $G = (A, B, E)$ is specified as algorithm **HDF** below. The *Sort-by-degree()* procedure sorts the vertices in order of descending degree. The "break" statement exits the smallest enclosing loop. The basic idea of **HDF** is give priority to coloring the vertices in order of their degree.

Algorithm HDF

Input: Bipartite graph $G = (A, B, E)$

Output: An edge-coloring of G

```
1. $\langle v_1, ..., v_n \rangle$ := Sort-by-degree($A \cup B$);
2. while $E \neq \{\}$
3.     $E' := \{\}$;
4.     for each $v$ read in sequence from $\langle v_1, ..., v_n \rangle$ {
5.         for each $e = (v, w) \in E$ {
6.             if $e$ is not adjacent to any edge in $E'$ {
7.                 Add $e$ to $E'$ and remove it from $E$
8.                 Reduce degree of $v, w$ by 1 and
                       remove from $\langle v_1, ..., v_n \rangle$
9.                 break
10.            }
11.        }
12.    }
13.    Color all edges in $E'$ with a new color
14.    $\langle v_1, ..., v_n \rangle$ := Sort-by-degree($A \cup B$);
15. }
16. end
```

Recall that the minimum number of colors, or schedule length, is d for a bipartite graph of degree d. How many colors will **HDF** use in the worst case? It is useful to answer a more general question instead: how many colors will a greedy heuristic use in the worst case? We first specify the *greedy heuristic* which, for every color, attempts to color as many edges as possible with that color.

Algorithm Greedy Heuristic

Input: Bipartite graph $G = (A, B, E)$

Output: An edge-coloring of G

```
1. Assign some order $F = \langle e_1, e_2, ..., e_m \rangle$ to the edges of $E$
2. i := 0
3. while $F \neq \{\}$ {
4.      for each $e$ read in sequence from $F$ {
5.           if $e$ can be colored with color $i$ {
6.                color $e$ with color $i$
7.                remove $e$ from $E$ and $F$
8.           }
9.      }
10.     i := i + 1
11. }
```

Clearly, any execution of **HDF** can be repeated by the greedy heuristic by choosing an appropriate initial ordering of the edges. Thus **HDF** is a special case of the greedy heuristic. We now state a simple but useful fact.

Lemma 4.1 *At the end of iteration i of the* **while** *loop of the greedy heuristic, if some vertex v is not fully colored and is not colored i, then all of v's partners are colored i.*

proof. Suppose not. Then v as well as at least one of its partners, say w, is not colored i. But then the greedy heuristic would have colored the edge (v, w) with i. □

Lemma 4.2 *The greedy heuristic produces a coloring using at most $2d - 1$ colors for a bipartite graph of degree d.*

proof. For a vertex v let $deg(v)$ be its degree and $P(v)$ its set of partners. Let $L(v) = deg(v) + \max\{deg(w) : w \in P(v)\}$. From Lemma 4.1, every color used by the greedy heuristic reduces $L(v)$ by at least 1. A special case occurs for the last color used to color v; for this case, there is one remaining edge incident on v, so that when it is colored $L(v)$ is reduced by 2. Therefore at most $L(v) - 1$ colors are used to color all edges incident to v. Since $L(v) \leq 2d$, the result follows. □

It can be shown that this bound is tight for **HDF**. A simple example where the $2d - 1$ bound on schedule length is met for $d = 2$ is the four-transfer example given in Chapter 1. However, we can prove that such an example bipartite graph $G(d)$ can be constructed for any positive integer d. In fact, we will show that $G(d)$ is a tree. We first introduce some notation and definitions; see Fig. 4.1 for examples of their use.

Notation. Upper-case italic letters denote vertices or subtrees of a tree; they may be subscripted. If A and B are vertices, $A;B$ denotes that they are siblings, and $A\langle B$ denotes that A is the parent of B. The letter R may be used to distinguish the root of a (sub)tree, and C for its child. Thus $R\langle C_1; C_2\rangle$, where the C_i are distinguished vertices, denotes a binary tree of two levels. A set of identical siblings is denoted using an array notation: thus $R\langle C[2]\rangle$ also denotes a binary tree with two levels. Angle brackets have higher precedence than semi-colons. Thus $R\langle C_1; C_2\rangle$; A denotes a forest with two trees, and $R\langle C_1; C_2; A\rangle$ denotes a ternary tree with two levels.

Def. Two trees S and T with roots R_S and R_T, respectively, are *root-merged* by deleting R_S and R_T (along with any incident edges), introducing a vertex R, and adding edges from R to every child of R_S and R_T. Using the notation above, and letting $+$ denote root-merging, let

$S = R_S\langle S_1; S_2; ...; S_i\rangle$

$T = R_T\langle T_1; T_2; ...; T_j\rangle$

Then $S + T = R\langle S_1; ...; S_i; T_1; ...; T_j\rangle$.

We now construct two families of trees to be used later in the construction of $G(d)$, and consider how they could be colored.

Def. The tree $F(i, d)$, with $d > 1$, is defined mutually recursively with the tree $H(i, d)$ as follows. See Fig. 4.1 for examples.

1. $F_{0,d}$ consists of a single vertex.

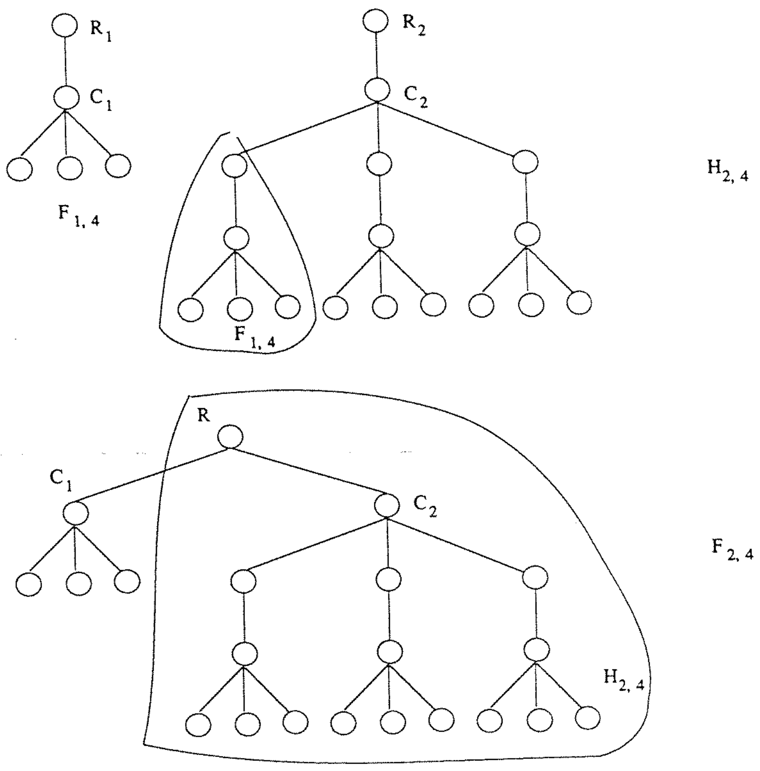

Figure 4.1: Example construction to show **HDF** takes up to $2d-1$ colors to color a graph of degree d

2. $H_{1,d} = R_1\langle C_1\langle F_{0,d}[d-1]\rangle$
3. $F_{1,d} = H(1,d)$
4. For $1 < i < d$, $H(i,d) = R_i\langle C_i\ \langle F_{i-1,d}[d-1]\rangle\rangle$
5. For $1 < i < d$, $F_{i,d} = H_{1,d} + H_{2,d} + \ldots + H_{i,d}$

$\langle\rangle$ has precedence over $+$, which has precedence over ;. Observe that for every tree $H_{i,d}$, the child of the root, C_i, has degree d, i.e., is critical. Also note that for every $F_{i,d}$, the children of its root are critical.

Lemma 4.3 *For every tree $F_{i,d}$, $0 < i < d$, there exists a sequence of choices* *made by* **HDF** *such that the root of* $F_{i,d}$ *is colored with every color in the set* $\{1, \ldots, i\}$.

proof. By induction over j.

base. $j = 1$. For $F_{1,d} = H_{1,d} = R_1\langle C_1\langle S[d-1]\rangle\rangle$, the choice of coloring edge (R_1, C_1) with color 1 suffices.

hyp. For all $F_{i,d}$, $0 < i < j < d$, there exists a sequence of choices made by **HDF** such that the root of $F_{i,d}$ is colored with all colors in {1, ..., i}.

ind. Consider the coloring of $F_{j,d}$. By definition,

$$F_{j,d} = H_{1,d} + H_{2,d} + \ldots + H_{j,d}$$

$$= R_j\langle C_1\langle F_{0,d}[d-1]\rangle;\ C_2\langle F_{1,d}[d-1]\rangle;\ \ldots\ C_j\langle F_{j-1,d}[d-1]\rangle;\ \rangle$$

We will show that there is a sequence of choices made by **HDF** such that for all $i, 0 < i \leq j$, edge (R_j, C_i) in the expression above is colored by color i. First note that by the hypothesis, for every i, $0 < i < j$, there exists a sequence of choices s_i such that the root of $F_{i,d}$ is colored with all colors in $\{1, \ldots, i\}$. Clearly, it is possible to merge these sequences appropriately so that the resulting sequence, s, colors the root of every $F_{i,d}$ with all colors in $\{1, \ldots, i\}$. We now show how s is extended by a sequence of choices that can be made by **HDF**.

Since every C_i is critical, and uncolored, it is eligible to be chosen by **HDF**. Let the first choice be to color C_1. By applying the hypothesis the root of every $F_{0,d}$, is not colored; let **HDF** choose to color C_1 by coloring (R_j, C_1) with the color 1. Now let **HDF** choose to color C_2. By the hypothesis, the root of every $F_{1,d}$ is colored with color 1; also, R_j was just colored 1. Therefore C_2 cannot be colored using color 1. Let **HDF** choose to color C_2 by coloring edge (R_j, C_2) with color 2.

HDF continues to choose to color each C_i in turn by choosing to color (R_j, C_i) with color i. The sequence of **HDF**'s choices is concatenated to the sequence s, and this gives the result. $\square$

Theorem 4.1 *For any positive integer d, there exists a bipartite graph of degree d and a sequence of choices made by* **HDF** *such that* **HDF** *uses $2d-1$ colors to color the graph.*

proof. By construction of the graph. Let

$$G(d) = R\langle F_{d-1,d}[d]\rangle$$

From Lemma 4.3 there exists a sequence of choices made by **HDF** such that the root of every $F_{d-1,d}$ is colored with all colors in $\{1, ..., d-1\}$. Therefore, each of the links incident to R will have to be colored with a color not in the set $\{1, ..., d-1\}$, and each will require a distinct color. Therefore d colors are required to color the links incident to R in addition to the colors $\{1, ..., d-1\}$,