An Asymptotically Optimal Algorithm for the Dutch National Flag Problem*

by

James R. Bitner
Department of Computer Science
University of Texas
Austin, Texas 78712

TR-118

November , 1979

*This work was supported in part by NSF grant MCS 77-02705

Abstract: We develop an algorithm for the Dutch National Flag problem that has an adjustable integer parameter smax ≥ 0 that allows a time-space trade-off. (Let n be the length of the input to be ordered.) The space required is proportional to smax (but independent of n), and the average number of swaps required is $\frac{6\ \text{smax} + 10}{18\ \text{smax} + 27}\, n + o(n)$. We show $1/3n + o(n)$ is a lower bound. Hence as smax $\rightarrow \infty$, the performance can be made arbitrarily close to the lower bound. We also study the problem of more than three colors.

Acknowledgement: I am grateful to Jay Misra for his helpful discussions on this problem.

1. Introduction

In the Dutch National Flag problem, we are given a sequence of n marbles, each of which is either red, white, or blue and we are to rearrange them such that all the red marbles occur first in the sequence, followed by all the whites, followed by all the blues. We are restricted to using the following two primitive functions in accessing the sequence: buck(i) which gives the color of the ith marble in the sequence, and swap(i,j), which interchanges the ith and jth marbles. Buck is deemed a very expensive operation and hence may be applied only once to each position in the sequence. Our objective is to find an algorithm to solve the problem which uses as few swaps as possible in the average case, where each of the 3^n initial sequences is equally likely. Also, the algorithm must operate using a constant amount of space, independent of the length of the sequence.

Notation: Through this paper n will denote the length of the sequence to be ordered and c will denote the number of possible colors (unless othewise noted, c=3).

This problem was initially posed and solved by Dijkstra [1], and later, another solution was obtained by Meyer [2]. McMaster [3] analyzed these solutions and found that Dijkstra's solution requires asymptotically $\frac{2}{3}$ n swaps on the average and Meyer's requires $\frac{5}{9}$ n. The main result of this paper is to develop a solution asymptotically* requiring $\frac{1}{3}$ n swaps, and further, prove this to be optimal.

The paper is organized as follows:

In section 2, we develop a correspondence between sequences of marbles and eulerian digraphs, which is then used to give a lower bound on the number

* – see below for a precise definition

of swaps required to order any given sequence for arbitrary c. We study the expected value of this lower bound to get a lower bound of $\frac{c-1}{2c}$ n on the average number of swaps required to order a sequence. In section 3 we study algorithms to solve the problem for arbitrary c which use more than a constant amount of space. One is the <u>shortest cycle first algorithm</u>, which for $c \leq 5$, orders any given sequence using the minimum number of swaps. For $c > 5$, we prove the <u>average</u> number of swaps used by this algorithm is asymptotically optimal as $n \to \infty$. We also study worst case bounds for $c > 5$.

In section 4, we use some ideas from the proof of the lower bound to develop an algorithm which solves the problem using constant space. The algorithm has a parameter, smax, that allows a time-space tradeoff. Increasing smax will reduce the number of swaps at the expense of requiring more space. It should be stressed that smax <u>does not</u> depend on n; therefore no matter what the value of smax, the algorithm still uses constant space (an amount proportional to smax). What we actually have is an infinite sequence of algorithms, each requiring constant space and each giving better and better performance. Finally, section 5 analyzes the algorithm. Asymptotically, for any smax ≥ 0, $\frac{6 \text{ smax} + 10}{18 \text{ smax} + 27}$ n swaps are required on the average. Even for smax = 0, the average of $\frac{10}{27}$ n swaps is better than Meyer's algorithm, and as smax $\to \infty$, the average number of swaps can be made arbitrarily close to the lower bound and, hence, is asymptotically optimal in some sense.

2. Lower Bounds

In this section we develop a correspondence between sequences of marbles and eulerian digraphs (see below for definition). We first use this correspondence to derive a lower bound on the number of swaps required to order any given sequence. We also use this correspondence to study the number of swaps required in the average case.

Definition: An eulerian digraph is a directed graph in which every vertex has its indegree equal to its outdegree. We allow a digraph to have multiple edges and self-loops. (Unless otherwise noted all our digraphs will be eulerian.) For an eulerian digraph G, let

e(G) be the number of edges in G, and

index(G) be e(G)-M(G) where M(G) is the number of cycles in a maximal decomposition of G into edge disjoint cycles. A cycle is a path in the digraph whose initial and final vertices are identical. It may pass through a vertex more than once. If a cycle passes through each vertex only once, it is a simple cycle. It is simplest to consider decompositions of digraphs into cycles and not require that the cycles be simple. (A maximal decomposition will, however, consist solely of simple cycles.)

To develop a correspondence between sequences of marbles and eulerian digraphs, we first divide a sequence into "regions." If there are a total of x_i marbles of color i in the sequence, let the first x_1 positions be region 1, the next x_2 in region 2, and so on. The digraph corresponding to this sequence has c vertices and is created by adding one edge from vertex i to vertex j for every marble of color i in region j. Note that a sequence is completely ordered iff for all i, all marbles of color i are in region i. Hence, the digraph corresponding to the completely ordered sequence consists solely of self-loops.

Theorem 2.1: A digraph constructed from a sequence as described above is an eulerian digraph.

Proof: For any i, the indegree of vertex i is the number of marbles of color i, and the outdegree is the number of marbles in region i. These quantities are equal by the definition of the regions. $\square$

To order a sequence, given any decomposition of the corresponding digraph (call it G) into edge-disjoint cycles*, we ignore the self loops and sequence through the remaining cycles in any order. If the current cycle is v_{i}, v_{i_2}, ..., v_{i_k}, there must be a marble of color i_j in region i_{j+1} for j=1, ..., k - 1 and a marble of color v_{i_k} in region v_{i_1}. Clearly k - 1 swaps can be used to put each of these k marbles in the correct region. Also note that after processing all the cycles, the sequence will be ordered. If the decomposition has k cycles with the ith having length L_i, the total number of swaps done is $\sum_{i=1}^{k}[L_i - 1] = e(G)-k$. Clearly, this quantity is minimized by using a maximal decomposition.

We now show that using the above procedure with the maximal decomposition uses the minimum number of swaps over all possible algorithms, not just those using this strategy. We use an "entropy argument", where index(G) is the entropy function. (G is the digraph corresponding to the current sequence in the execution of some algorithm.) Index(G) must be decreased using swaps from its initial value down to zero. (A sequence is ordered iff every marble is in the correct region, i.e. its corresponding digraph has index equal to zero.) The following lemma shows that a swap cannot decrease index(G) by very much.

Lemma 2.1: Let S be any sequence and S' be any sequence obtained from S

*It is easy to show that every eulerian digraph has such a decomposition, see for example [4].

by doing one swap. Let G and G' be the digraphs corresponding to, respectively, S and S'. Then $\text{index}(G') \geq \text{index}(G)-1$.

Proof: The effect on G of swapping a color i marble in region j with a color k marble in region ℓ is shown in Figure 2.1. (i, j, k, and ℓ are not necessarily distinct.) Edges e_1 and e_2 in G are replaced by e_1' and e_2' to form G'. Since $e(G)=e(G')$, proving $M(G') \leq M(G)+1$ will prove the theorem. Suppose $M(G') > M(G) + 1$ and let a maximal decomposition of G' be $C_1, \ldots, C_m$ (Where $m = M(G')$).

We consider two cases:

Case 1 (e_1' and e_2' are on the same C_h):

Let $P_{\ell k}$ be the portion of C_h from ℓ to k and P_{ji} the portion from j to i. Form two cycles in G: C_h', which consists of e_1 and P_{ji}, and C_h'', which consists of e_2 and $P_{\ell k}$. Hence $C_1, \ldots, C_{h-1}, C_h', C_h'', C_{h+1}, \ldots, C_m$ is a decomposition of G into $m+1$ cycles, a contradiction.

Case 2 (e_1' and e_2' are on different C_h's, say, C_1 and C_2):

Let $P_{\ell i}$ be the portion of C_1 from ℓ to i and P_{jk} be the portion of C_2 from j to k. Form a cycle C_1' in G consisting of $e_1, P_{jk}, e_2, P_{\ell i}$. Then C_1', $C_3, \ldots, C_m$ forms a decomposition of G into $m-1$ cycles, again, a contradiction.

Hence $M(G') \leq M(G) + 1$ and the lemma is proved. $\square$

Theorem 2.2: Given a sequence S_o, let G_o be the corresponding digraph. Then at least $\text{index}(G_o)$ swaps must be used in ordering S_o.

Proof: Suppose k swaps are used. Let S_i be the sequence after i swaps and G_i be the corresponding digraph. Since S_k is ordered, G_k consists solely of self-loops, and $\text{index}(G_k) = 0$.

By Lemma 2.1, $\text{index}(G_i) \geq \text{index}(G_{i-i}) - 1$ and clearly $k \geq \text{index}(G_o)$. $\square$

Corollary 2.1: Using a maximal decomposition in the manner previously described orders any given sequence in the minimal number of swaps.

Theorem 2.2 provides a lower bound and Corollary 2.1 shows it is achievable (though not necessarily by an algorithm using constant space.) We will discuss the problem of actually finding a maximal decomposition in the next section.

Before considering the average value of index(G) in order to obtain a lower bound on the average number of swaps, we introduce some notation and prove two lemmas.

Definition: For any n and $1 \le i, j \le c$ let the random variable $I_{ij}^{(n)}$ be the number of edges from i to j in the digraph corresponding to a random arrangement of n marbles with c possible colors. (Note that $I_{ij}^{(n)}$ also depends on c, but we omit this to simplify the notation.)

Definition: By the phrase "number of 2-cycles in a digraph," we mean the maximum number of 2-cycles possible in any decomposition. It is easy to see that this quantity can be found by successively choosing any 2-cycle until more remain,(we can never make a "bad" choice)and hence is given by

$$\sum_{i<j} \min (I_{ij}^{(n)}, I_{ji}^{(n)})$$

Lemma 2.2: Let $A_1, A_2, \ldots$ and $B_1, B_2, \ldots$ be sequences of random variables such that for any $\varepsilon > 0$,

$$\text{Prob} \left(\left| \frac{A_n}{n} - L \right| < \varepsilon \right) \to 1 \text{ and Prob} \left(\left| \frac{B_n}{n} - M \right| < \varepsilon \right) \to 1$$

then

$$\text{Prob} \left(\left| \frac{A_n}{n} - L \right| < \varepsilon \text{ and } \left| \frac{B_n}{n} - M \right| < \varepsilon \right) \to 1$$

Proof: Given any $\varepsilon > 0$, we need to show that for all $\delta > 0$ there exists an $N_{\delta,\varepsilon}$, such that for all $n > N_{\delta,\varepsilon}$,

$$\text{Prob}\ (|\frac{A_n}{n} - L| < \varepsilon \text{ and } |\frac{B_n}{n} - M| < \varepsilon) > 1 - \delta.$$

We can choose $N_{\delta,E}$ large enough so that

$$\text{Prob}\ (|\frac{A_n}{n} - L| < \varepsilon) > 1 - \frac{\delta}{2} \quad \text{and Prob}\ (|\frac{B_n}{n} - M| < \varepsilon) > 1 - \frac{\delta}{2}$$

for all $n > N_{\delta,\varepsilon}$. Then

$$\text{Prob}\ (|\frac{A_n}{n} - L| < \varepsilon \text{ and } |\frac{B_n}{n} - M| < \varepsilon) =$$

$$\text{Prob}\ (|\frac{A_n}{n} - L| < \varepsilon) + \text{Prob}\ (|\frac{B_n}{n} - M| < \varepsilon) - \text{Prob}\ (|\frac{A_n}{n} - L| < \varepsilon \text{ or}$$
$$|\frac{}{n} - M| < \varepsilon) > 1 - \delta$$

since the last probability is at most one. $\square$

Lemma 2.3: Given any eulerian digraph, there exists a maximal decomposition containing all the self-loops and 2-cycles.

Proof: Clearly, each self-loop must occur in any maximal decomposition. Suppose, however, that some 2-cycle does not. Choose any maximal decomposition and let C_1 and C_2 be the two cycles which contain the edges of the 2-cycle (see Figure 2.2). Then a new decomposition can be formed by replacing C_1 and C_2 by the 2-cycle and the cycle consisting fo the remaining edges in C_1 and C_2. Since this decomposition has at least as many cycles as the original, it too is maximal, a contradiction.

The average case analysis relies heavily on the law of large numbers. For three colors, we expect, with high probability, a nearly equal number of red, white and blue marbles. Hence, we expect the three regions to be of nearly equal length and to contain approximately the same number of red, white and blue marbles. This results in a digraph consisting nearly exclusively of self-loops and 2-cycles. Lemma 2.3 can then be used to give a nearly complete decomposition of the digraph. This intuitive explanation is formalized in the proof of the following theorem.

Theorem 2.3: Asymptotically, the average number of swaps required to order a sequence of three colors is at least 1/3 $n + o(n)$.

Proof: By Theorem 2.2, the minimum number of swaps required by an algorithm to order a given sequence is $e(G) - M(G)$ where G is the corresponding digraph. By Lemma 2.3 $M(G) = S + T + U$

where S is the number of self-loops. T is the number of 2-cycles in G, and, as noted previously,

$$T = \min (I_{12}^{(n)}, I_{21}^{(n)}) + \min (I_{13}^{(n)}, I_{31}^{(n)}) + \min (I_{23}^{(n)} + I_{32}^{(n)}).$$

U is the number of cycles remaining after the self-loops and 2-cycles have been removed. Clearly,

$$U = \max (I_{12}^{(n)}, I_{21}^{(n)}) - \min (I_{12}^{(n)}, I_{21}^{(n)}).$$

Since $e(G) = S + 2T + 3U$, we have $e(G) - M(G) = T + 2U$, and $E(e(G) - M(G))$

$= E(\min (I_{12}^{(n)}, I_{21}^{(n)}) + \min (I_{13}^{(n)}, I_{31}^{(n)}) + \min (I_{23}^{(n)}, I_{32}^{(n)}) + 2 (\max (I_{12}^{(n)}, I_{21}^{(n)}) - \min (I_{12}^{(n)}, I_{21}^{(n)})))$

$$= E (\min (I_{12}^{(n)}, I_{21}^{(n)})) + 2 \cdot E(\max (I_{12}^{(n)}, I_{21}^{(n)})) \tag{1}$$

Claim 1: For any $\varepsilon > 0$, Prob $(|\frac{I_{12}^{(n)}}{n} - \frac{1}{9}| < \varepsilon) \to 1$

Proof: Define the following random variables:

$J_{12}^{(n)}$ as the number of marbles of color 1 in positions $\frac{n}{3}$ through $\frac{2n}{3}$.

$B_1^{(n)}$ as the total number of marbles of color 1,

$B_2^{(n)}$ as the total number of marbles of color 1 or 2,

$U_1^{(n)} = |B_1^{(n)} - \frac{n}{3}|$ and, $U_2^{(n)} = |B_2^{(n)} - \frac{2n}{3}|$.

$J_{12}^{(n)}$ is our estimate of $I_{12}^{(n)}$. $U_1^{(n)}$ and $U_2^{(n)}$ measure how near the boundaries of region 2 ($B_1^{(n)}$ and $B_2^{(n)}$) are to their means, giving a measure of how accurately $J_{12}^{(n)}$ measures $I_{12}^{(n)}$. Choose any $\varepsilon > 0$. By the Law of Large Numbers,

$$\text{Prob } (|\frac{J_{12}^{(n)}}{n} - \frac{1}{9}| < \varepsilon) \to 1 \tag{2}$$

$$\text{Prob } \left(\left|\frac{B_1^{(n)}}{n} - \frac{1}{3}\right| < \varepsilon\right) \to 1 \text{ so Prob } \left(\frac{U_1^{(n)}}{n} < \varepsilon\right) \to 1 \qquad (3)$$

$$\text{Prob } \left(\left|\frac{B_2^{(n)}}{n} - \frac{2}{3}\right| < \varepsilon\right) \to 1 \text{ so Prob } \left(\frac{U_2^{(n)}}{n} < \varepsilon\right) \to 1 \qquad (4)$$

We show

$$\text{Prob } \left(\left|\frac{I_{12}^{(n)}}{n} - \frac{1}{9}\right| < \varepsilon\right) \geq \text{Prob } \left(\left|\frac{J_{12}^{(n)}}{n} - \frac{1}{9}\right| < \frac{\varepsilon}{2} \text{ and } \frac{U_1^{(n)}}{n} < \frac{\varepsilon}{4} \text{ and } \frac{U_2^{(n)}}{n} < \frac{\varepsilon}{4}\right) \qquad (5)$$

by demonstrating that every sequence satisfying the condition of the second probability also satisfies that of the first. We have the bound

$$J_{12}^{(n)} - U_1^{(n)} - U_2^{(n)} \leq I_{12}^{(n)} \leq J_{12}^{(n)} + U_1^{(n)} + U_2^{(n)}$$

or, equivalently,

$$\frac{J_{12}^{(n)}}{n} - \frac{1}{9} - \frac{U_1^{(n)}}{n} - \frac{U_2^{(n)}}{n} \leq \frac{I_{12}^{(n)}}{n} - \frac{1}{9} \leq \frac{J_{12}^{(n)}}{n} - \frac{1}{9} + \frac{U_1^{(n)}}{n} + \frac{U_2^{(n)}}{n} \qquad (6)$$

Then if a sequence satisfies the condition of the second probability, (6) becomes

$$-\varepsilon \leq \frac{I_{12}^{(n)}}{n} - \frac{1}{9} \leq \varepsilon \qquad (7)$$

and hence the sequence also satisfies the condition of the first probability. But from (2), (3), and (4) and Lemma 2.2, the right-hand side of (5) approaches one as $n \to \infty$. Hence the left-hand side must also, proving the claim.

Claim 2: For any $\varepsilon > 0$, Prob $\left(\left|\frac{I_{21}^{(n)}}{n} - \frac{1}{9}\right| < \varepsilon\right) \to 1$

Proof: Similar to Claim 1

Claim 3: For any $\varepsilon > 0$, Prob $\left(\left|\frac{I_{12}^{(n)}}{n} - \frac{1}{9}\right| < \varepsilon \text{ and } \left|\frac{I_{21}^{(n)}}{n} - \frac{1}{9}\right| < \varepsilon\right) \to 1$

Proof: By Claims 1 and 2 and Lemma 2.2

Claim 4: $E\left(\min\left(\frac{I_{12}^{(n)}}{n}, \frac{I_{21}^{(n)}}{n}\right)\right) \to \frac{1}{9}$

Proof: Choose some $\varepsilon > 0$. F or $n \geq 0$ let $A^{(n)}$ be the event "$\left|\frac{I_{12}^{(n)}}{n} - \frac{1}{9}\right| < \varepsilon$ and $\left|\frac{I_{21}^{(n)}}{n} - \frac{1}{9}\right| < \varepsilon$".

then

$$E\left(\min\left(\frac{I_{12}^{(n)}}{n}, \frac{I_{21}^{(n)}}{n}\right)\right) = \text{Prob }(A^{(n)}) \cdot E\left(\min\left(\frac{I_{12}^{(n)}}{n}, \frac{I_{21}^{(n)}}{n}\right) \middle| A^{(n)}\right) + \text{Prob }(\bar{A}^{(n)}) \cdot E\left(\min\left(\frac{I_{12}^{(n)}}{n}, \frac{I_{21}^{(n)}}{n}\right) \middle| \bar{A}^{(n)}\right).$$

By Claim 3, Prob $(A^{(n)}) \to 1$. Hence

$$E\left(\min\left(\frac{I_{12}^{(n)}}{n}, \frac{I_{21}^{(n)}}{n}\right)\right) \to E\left(\min\left(\frac{I_{12}^{(n)}}{n}, \frac{I_{21}^{(n)}}{n}\right) \middle| A^{(n)}\right)$$

But given event $A^{(n)}$ occurs.

$$\frac{1}{9} - \varepsilon \leq \frac{I_{12}^{(n)}}{n}, \frac{I_{21}^{(n)}}{n} \leq \frac{1}{9} + \varepsilon$$

by the definition of $A^{(n)}$. Hence

$$\frac{1}{9} - \varepsilon \leq E\left(\min\left(\frac{I_{12}^{(n)}}{n}, \frac{I_{21}^{(n)}}{n}\right) \middle| A^{(n)}\right) \leq \frac{1}{9} + \varepsilon$$

for any $\varepsilon > 0$, proving $E\left(\min\left(\frac{I_{12}^{(n)}}{n}, \frac{I_{21}^{(n)}}{n}\right)\right) \to \frac{1}{9}$.

Claim 5: $E\left(\max\left(\frac{I_{12}^{(n)}}{n}, \frac{I_{21}^{(n)}}{n}\right)\right) \to \frac{1}{9}$

Proof: The proof of Claim 4 also holds if "min" is replaced by "max".

Finally, substituting Claims 4 and 5 into (1) proves the theorem. $\square$

Finally, we extend the result to an arbitrary number of colors.

Lemma 2.4: For any $c \geq 1$, let S, T and U be, respectively, the number of self-loops, 2-cycles and other cycles in a maximal decomposition of a digraph found by first removing all self-loop and 2-cycles. Then

$$E(S) = \frac{n}{c} + o(n),\ E(T) = \frac{(c-1)n}{2c} + o(n),\ E(U) = o(n).$$

Proof: The number of self-loops at vertex i is $I_{ii}^{(n)}$ and the number of 2-cycles between vertices i and j is $\min(I_{ij}^{(n)}, I_{ji}^{(n)})$. In a manner similar to Theorem 2.3 we can show

$$E\left(\frac{I_{ii}^{(n)}}{n}\right) \to \frac{1}{c^2} \text{ and } E(\min(I_{ij}^{(n)}, I_{ji}^{(n)})) \to \frac{1}{c^2}.$$

Thus, there will be $E(I_{ii}^{(n)}) = \frac{n}{c^2} + o(n)$ self loops at vertex i (for a total of

$\frac{n}{c} + o(n))$, and $E(\min (I_{ij}^{(n)}, I_{ji}^{(n)})) = \frac{n}{c^2} + o(n)$ 2-cycles from i to j

(for a total of $\frac{(c-1)\ n}{2c} + o(n))$.

By Lemma 2.3, we can choose these cycles first and still get a maximal decomposition. These cycles account for $n + o(n)$ edges, leaving at most $o(n)$ edges for the remaining cycles. $\square$

Theorem 2.4: The average number of swaps required to order an arrangement with c colors is $\frac{c-1}{2c}\ n + o(n)$

Proof: Let S, T, and U be as in Lemma 2.4, and let M be the number of swaps resulting from this maximal decomposition. Then $0 \cdot S + 1 \cdot T \leq M \leq 0 \cdot S + 1 \cdot T + c \cdot U$. By Lemma 2.4, $E(M) = \frac{(c-1)}{2c}\ n + o(n)$. $\square$

3. Algorithms for Finding a Maximal Decomposition

In this section, we discuss algorithms to solve the Dutch National Flag Problem where an arbitrary number of colors are allowed. We will drop the constant space assumption and study algorithms that first count the number of marbles of each color, then construct the digraph associated with the given sequence and decompose it in some manner into cycles, then sequence through the cycles and perform the appropriate sequence of swaps to "remove" each cycle. Hence, the problem reduces to one of finding a maximal or near-maximal decomposition. Though finding a maximal decomposition for arbitrary c appears to be a hard problem, we present a simple, "greedy" algorithm, the shortest cycle first algorithm, which, at each step, arbitrarily removes any of the shortest cycles remaining in the digraph. (The algorithm can either be thought of as non-deterministic, or as specifying a class of algorithms.) We prove that if the digraph has at most five vertices (i.e. $c \leq 5$), the shortest cycle first algorithm finds a maximal decomposition. If $6 \leq c \leq 8$, the shortest cycle first algorithm might find a maximal decomposition (i.e. some "shortest cycle first" decomposition exists for every digraph.) We prove a very general class of algorithms gives an asymptotically optimal decomposition in the average case and conclude with a worst case bound on the performance of this class.

Theorem 3.1: For an eulerian digraph with at most five vertices (i.e. $c \leq 5$) the shortest cycle first algorithm finds a maximal decomposition.

Proof: Consider any digraph G. If $M(G) = 0$, the shortest cycle first algorithm will trivially be successful. We proceed by induction on $M(G)$. Suppose C is the first cycle chosen by the shortest cycle first algorithm and let x be the length of C.

We first show that M(G–C) = M(G) - 1. (i.e. C was not a bad first choice.) This is trivially true if C is contained in some maximal decomposition, so suppose no maximal decomposition contains C. Choose any one. Though it does not contain C, every edge in C must be included in one of its cycles. Suppose y cycles in this decomposition have edges in common with C. Consider a new decomposition (see Figure 3.1) where we choose C, then the long cycle consisting of the remainder of the y cycles (call this C') and then choose the remainder of the original decomposition. Since each of the y original cycles has length at least x (the length of the shortest cycle), C' must have length at least xy - x (x must be subtracted because C has been removed.) Hence there must be a vertex occurring at least $\left\lceil \frac{xy - x}{c} \right\rceil$ times on C', and C' can be split into at least $\left\lceil \frac{xy - x}{c} \right\rceil$ simple cycles. The new decomposition has at least $1 + \left\lceil \frac{xy - x}{c} \right\rceil$ cycles in place of the y of the original decomposition. Since it is easy to verify that $1 + \left\lceil \frac{xy - x}{c} \right\rceil \geq y$ for $2 \leq y \leq x \leq n \leq 5$, the new decomposition has at least as many cycles as the original, providing a maximal decomposition containing C, a contradiction. Hence M(G - C) = M(G) - 1.

By induction, applying the shortest cycle first algorithm to G - C will correctly give a decomposition into M(G - C) = M(G) - 1 cycles. This, added to C gives a decomposition into M(G) cycles for G, proving the theorem.

If n = 6, more care is required. If we arbitrarily choose from among the shortest cycles, an optimal decomposition might not result. (See Figure 3.2). However, we can prove the following, weaker, result.

Theorem 3.2: Given any eulerian digraph with 6, 7, or 8 vertices, there exists a maximal decomposition whose cycles can be ordered, such that, if the cycles are removed in that order from the digraph, a shortest cycle is removed at each step. (i.e. there is some "shortest cycle first" decomposition.)

Proof: (Similar to Theorem 3.1). Consider one of the shortest cycles in a given digraph. Call it C and suppose it has x nodes. If C does not occur in any maximal decomposition, consider the y cycles that intersect C. If any has length x, it can be chosen, and the theorem can be proven by induction as Theorem 3.1. If none do, then we proceed as in Theorem 3.1, forming C', which now has y(x+1)-x vertices and must form $\left\lceil \frac{y(x+1)-x}{c} \right\rceil$ cycles. Verifying that
$1 + \left\lceil \frac{y(x+1)-x}{c} \right\rceil \geq y$ for $2 \leq y \leq x \leq c \leq 7$
proves the theorem for $c = 6$ and 7.

The above proof "almost works" for $c = 8$; the inequality is violated only when $x = y = 4$. We analyze this situation as a special case. Suppose the theorem is false. Then there exists a digraph (G) whose shortest cycle, c (v_1, v_2, v_3, v_4), has length 4, and C is not in any maximal decomposition. Further, since $y = 4$, each edge of C is in a different cycle in the optimal decomposition. The length of each of these four cycles is at least 5.

Consider the graph (G') consisting of these four cycles with the edges in C removed. Note that G' is a cycle. We repeatedly use the fact that G' can be partitioned into at most 2 cycles. (Otherwise its decomposition plus C would give at least 4 cycles and choosing C would provide an maximal decomposition.) We number the vertices in G' $x_1, x_2, \ldots, x_k$ by tracing the path from v_1 to v_4 to v_3 to v_2 to v_1. (Since $c = 8$ the x's are not all distinct.) Since each of the original cycles intersecting C had length at least 5, $k \geq 16$. However, we cannot have $k > 16$ because then a vertex would occur at least three times in

x_1, ..., x_k, providing a decomposition of G' into at least three cycles. Also, there cannot be a cycle in G' of length less than 8. Otherwise, removing this cycle would result in a cycle with at least 9 vertices. Hence one must occur twice, giving two cycles for a total of three.

Therefore, G' consists of two cycles of length 8; its form must be as shown in Figure 3.3. Assume without loss of generality that v_1 is the vertex marked x. Then v_4 must be the vertex marked y since it is at distance 4 from v_1 along the cycle, and v_3 must be the vertex marked x since it is at distance from v_4. However, this implies $v_1 = v_3$, a contradiction. Hence, the conjectured digraph cannot exist, and the theorem is proved. $\square$

For $c \geq 9$, no such theorem can be proved. Figure 3.4 provides a digraph for which no shortest cycle first decomposition exists.

If we consider the average case, the analysis is especially simple since the associated digraph is nearly all 2-cycles. We consider the <u>all 2-cycles first algorithm</u>, which first removes all self-loops and 2-cycles then <u>arbitrarily</u> decomposes the digraph.

<u>Theorem 3.3</u>: The all 2-cycles first algorithm is asymptotically optimal in the average case.

<u>Proof</u>: Immediate from Lemma 2.3 and 2.4

<u>Corollary 3.1</u>: The shortest cycle first algorithm is asymptotically optimal in the average case.

We now consider worst case performance. Rather than study the size of the decomposition, we study the number of swaps required in ordering a sequence based on this decomposition (i.e. $e(G) - k$ where k is the number of cycles in the decomposition.)

Theorem 3.4: Given any sequence, let COST be the number of swaps required using the all 2-cycles first algorithm and let COST_{OPT} be the optional number of swaps, then $\frac{\text{COST}}{\text{COST}_{\text{OPT}}} \leq \frac{3}{2}$. Further, $\frac{3}{2}$ is the smallest possible constant bound.

Proof: Let S and T be the number of self-loops and 2-cycles in the digraph. The all 2-cycles first algorithm results in a decomposition of at least $S + T$ cycles. Hence $\text{COST} \leq n - S - T$. Using Lemma 2.3, there is a maximal decomposition with $S + T + U$ cycles, where U is the number of cycles of length greater than two. Clearly, $U \leq \frac{n - S - T}{3}$ and $\text{COST}_{\text{OPT}} = n - S - T - U \leq n - S - T - \frac{n - S - T}{3} = \frac{2}{3}[n - S - T]$. Hence $\frac{\text{COST}}{\text{COST}_{\text{OPT}}} \leq \frac{3}{2}$.

To prove this bound is tight, consider the eulerian digraph in Figure 3.5, an n-cycle where each edge of the n-cycle is the base of a triangle. The digraph has 3n edges, and the optimal decomposition is clearly the n triangles. However, the all 2-cycles first algorithm might decompose the digraph into two cycles, the n-cycle and the cycle consisting of the remaining 2n edges. Hence $\frac{\text{COST}}{\text{COST}_{\text{OPT}}} = \frac{3n-2}{3n-n}$, which approaches $\frac{3}{2}$ as $n \to \infty$. $\square$

This bound does not appear to be tight for the shortest-cycle first algorithm. The next theorem gives the worst example found so far, which would require a bound of $\frac{5}{4}$.

<u>Theorem 3.5</u>: Let COST_{SCF} be the number of swaps using the shortest cycle first algorithm. Then for any $n \geq 1$ there is an eulerian digraph (and hence a sequence) where $\frac{\text{COST}_{\text{SCF}}}{\text{COST}_{\text{OPT}}} = \frac{5n-3}{4n-2}$.

<u>Proof</u>: Consider the eulerian digraph in Figure 3.6, which has $6n-3$ edges. Its maximal decomposition is into the $2n-1$ triangles pointing down. However, a possible shortest cycle first decomposition is into the $n-1$ triangles pointing up, and the long cycle along the perimeter. In this case,

$$\frac{\text{COST}_{\text{SCF}}}{\text{COST}_{\text{OPT}}} = \frac{6n-3-n}{6n-3-(2n-1)} = \frac{5n-3}{4n-2}. \qquad \square$$

4. A Constant Space Algorithm

In this section we describe a constant space algorithm to solve the Dutch National Flag problem. The algorithm has an adjustible integer parameter smax, which may be any non-negative integer. For any value of smax, the algorithm uses space proportional to smax (and independent of n). We show in the next section that a time/space tradeoff results; as smax $\rightarrow \infty$, the average number of swaps required $\rightarrow$ 1/3n, the optimum. Since we can get arbitrarily close to the lower bound by choosing smax sufficiently large, the algorithm is "asymptotically optimal" in some sense.

The algorithm has two phases. In the first, three pointers, r, w, and b are used to delimit regions consisting solely of, respectively, red, white, and blue marbles according to the invariant:

"($1 \leq i \leq r$ -> buck(i) = red) and
($n/3 \leq i \leq w$ -> buck(i) = white) and
($2n/3 \leq i < b$ -> buck(i) = blue) and
buck(r) $\neq$ red and buck(w) $\neq$ white and buck(b) $\neq$ blue."

Initially, r=0, w=n/3-1, and b=2n/3-1. Then r is successively incremented until buck(r) $\neq$ red. (We refer to this operation as "advancing r".) w and b are advanced in a similar manner. The algorithm then iterates, advancing the pointers and performing swaps when necessary to preserve the invariant. The first phase ends when r=n/3+1 or w=2n/3+1 or b=n+1. The second phase merges the unexamined regions (from r to n/3-1, w to 2n/3-1 and b to n) into the red, white, and blue regions, and possibly shifts the white region left or right. By the analysis in Theorem 2.3, the average size of the unexamined regions and the average distance the white region must be shifted is o(n). Thus, any algorithm (such as a straightforward extention of Dijkstra's [1] algorithm or Meyer's algorithm [2]) can be used in the second phase without affecting the asymptotic performance of the algorithm. Hence, for simplicity, we examine only the first phase. Another simplification is to ignore the termination condition, some tests to keep $r \leq n/3$ and $w \leq 2n/3$ (needed so that buck is not called twice), and a test to keep $b \leq n$ (so that b remains a valid subscript). These would add simple, but obscuring details to the discussion.

Two important ideas are used to reduce the number of swaps. First, w is initially n/3, not n as in the algorithms of Dijkstra and Meyer. Hence the white region will be from n/3 to w instead of from w to b-1. By Theorem 2.3, the final positions of the white marbles will, with probability approaching one, be approximately from n/3 to 2n/3. Hence white marbles moved into the white regions have, most likely, come to their final resting place.

Before discussing the second idea, we describe the only two situations in which the algorithm swaps marbles. If buck(r) = white and buck(w) = red, then swap(r,w) is called a "single swap" (and similarly for the other two pairs of pointers). On the other hand, if buck(r) = white, buck(w) = blue, and buck(b) = red, then the sequence swap(r,w); swap(w,b) is called a "double swap" (and similarly if buck(r) = blue). The second idea then is to avoid double swaps as much as possible, preferring a single swap, which puts two marbles in place at the cost of one swap, to a double swap, which puts three marbles in place at the cost of two swaps. Clearly, this is motivated by the idea of finding a maximal decomposition.

To accomplish this, a single swap is done on each iteration, if one is possible. Otherwise, a "scout" is advanced from r in an attempt to find a marble that will permit a single swap. (This scout is referred to as the "lead scout".) If such a marble is found, a single swap with the lead scout will be performed on the next iteration. Otherwise, each successive iteration will advance a scout from the lead scout (with this new scout becoming the lead scout), until either a single swap becomes possible, or the maximum number of scouts has been reached and there is no alternative to doing a double swap. Now, instead of preserving the relation "$(1 \leq i < r) \rightarrow$ buck(i) = red" we preserve "$((1 \leq i < \ell s)$ and i is not a scout) $\rightarrow$ buck(i) = red)" where ℓs is the lead scout. Using the scouts in this manner avoids multiple calls to buck.

The preceeding gives the general strategy of the algorithm (which can be found in the Appendix). To aid in understanding the program a brief, informal description of each procedure follows. This is not to imply the program cannot be formally verified; the reader can verify the program from the pre- and post-conditions given for each procedure.

The following terminology is used: "smax" is the maximum number of scouts, and "s" is the number of currently active scouts. The scouts are kept in an array "scout", and "scout[s]" is the lead scout. It is convenient to store the value of "r" in scout[0]. Finally, rcolor, scolor, wcolor, and bcolor record the color of the marble pointed to by, respectively, r, the lead scout, w and b.

Function swap_possible returns true iff a single swap is possible. All four of r, the lead scout, w, and b are considered in this determination. Procedure advance advances a pointer over marbles of a given color and returns the color of the first marble encountered which is not of that color.

Procedure single_swap performs a single swap, given that one is possible. Since it could involve either r or the lead scout, a test is made at the beginning of the procedure to determine which, if either, is involved. The lead scout is used unless it has already been swapped and hence scolor = red. (Note that if $s \neq 0$ and scolor = rcolor, either r or the lead scout could be swapped. However, the algorithm is simpler if we choose to use the lead scout in this case. If $s = 0$ they are identical, so it makes no difference.) The procedure then determines which pair of marbles is to be swapped, swaps them, and advances the appropriate pointers. Procedure double_swap is similar.

Advance_r advances the r pointer and the scouts immediately after a red marble has been swapped. There are two cases: if scolor $\neq$ red, the lead scout was swapped and all that is needed is to record this fact by setting scolor := red. Otherwise, r was swapped. Since buck(r) = red, r provides no useful information. If s $\neq$ 0, each scout (including r, which is scout[0]) can be "advanced" by setting scout[i] := scout[i+1] for i = 1, ..., s-1. Scout[s] is no longer needed and is discarded by setting s := s-1. If s is now zero, r must be advanced to the next non-red marble by calling advance.

Procedure advance_scouts advances a scout from the lead scout. If scolor = red, the lead scout itself can be advanced, since buck(scout[s]) = red, and hence scout[s] contains no useful information. Otherwise a new scout is advanced.

5. Analysis of the Algorithm

The algorithm is analyzed in the following theorem.

Theorem 5.1: The average number of swaps required by the algorithm of Section 4 is $\frac{6\ \text{smax} + 10}{18\ \text{smax} + 27}\ n + o(n)$ for any smax ≥ 0.

Proof: We consider only smax > 0. For smax = 0 the Markov chain we will construct degenerates to two states. With the techniques used below, it is easily shown the algorithm will require $\frac{10}{27}\ n + o(n)$ swaps, and hence the theorem is true in this case.

Definition: The predicate hascolors(x, y, z) is defined to be true iff (rcolor = x) and (wcolor = y) and (bcolor = z).

To model the algorithm by a Markov chain, we examine the program variables each time control reaches the top of the while loop in the main program. The values of the variables determine the current state (see Figure 5.1). We briefly discuss the motivation for these definitions. To predict the behavior of the algorithm, we must know the values of rcolor, wcolor and bcolor. (Note that for each triple of values with rcolor = white, there is a corresponding "equivalent" triple with rcolor = blue. Thus the eight possible triples result in only four types of states.) In addition, if hascolors(white, blue, red) or hascolors(blue, red, white), we must know whether the lead scout can be swapped (i.e. scolor ≠ red and scolor ≠ rcolor) since this determines whether a swap is possible. Finally, we need ℓ defined by

$$\ell = \begin{cases} s \text{ if scolor} \neq \text{red} \\ s-1 \text{ if scolor} = \text{red} \end{cases}$$

(Note that for analysis purposes s scouts when the lead scout has been swapped (i.e. scolor = red) is equivalent to s-1 scouts when the lead scout has not been swapped. In both cases, there will be s scouts after a call to advance_r.) As a simplification, we note that all states satisfying $\ell = 0$ and swap_possible are equivalent and these are merged to form only one state.

The state transition probabilities (shown in Figure 5.2) can be calculated from several simple observations. In states D_i, E_i, and F_i, swap_possible is true, and a single swap is done. If w is swapped, wcolor becomes <u>arbitrary</u> (by this we mean wcolor = red with probability 1/2 and wcolor = blue with probability 1/2). Similarly, if b is swapped, bcolor becomes arbitrary (i.e., equal to red or white, either with probability 1/2). If r or the lead scout is swapped (it makes no difference which for purposes of analysis), rcolor is unchanged and ℓ is decremented. A single swap is also performed in state A, but here the colors of both the pointers swapped (including r) become arbitrary.

In state B_i, $0 \le i \le$ smax − 1, swap_possible is false, and advance_scout is called. With probability 1/2, it is successful (i.e. afterwards scolor ≠ rcolor and scolor ≠ red). A single swap is now possible, and a transition to state C_{i+1} results. Otherwise, a transition to B_{i+1} results. In state B_{smax}, double_swap is called, resulting in wcolor and bcolor becoming arbitrary (though rcolor remains the same), and ℓ being decremented by one. Finally, in state C_i, $1 \le i \le$ smax, we have scolor ≠ rcolor and scolor ≠ red. This results in the lead scout being swapped. Scolor becomes red, and hence ℓ is decremented, and the color of the pointer with which the lead scout was swapped becomes arbitrary.

A system of equations can be constructed to determine the steady state probabilities (see Figure 5.3). We use lower case letters to denote the probability of the state named by the same upper case letter. Solving it, we get:

$$a = \frac{5}{5\ smax+4}$$

$$b_0 = \ldots = b_{smax-1} = \frac{2}{5\ smax + 4}$$

$$c_1 = \dots = c_{smax} = d_1 = \dots = d_{smax-1} = b_{smax} = \frac{1}{5\ smax + 4}$$

$$e_1 = \dots = e_{smax-1} = f_1 = \dots = f_{smax-1} = \frac{1}{2(5\ smax + 4)}$$

From these probabilities, we determine the expected number of swaps. Define the following random variables: Let S be the total number of swaps performed by the algorithm, and let S_1 and S_2 be, respectively, the number of single and double swaps performed by Phase 1, and S_3 be the number of swaps required in Phase 2. Let R be the sum of the lengths of the red, white and blue regions at the end of Phase 1, that is, $R = r + (w - \frac{n}{3}) + (b - \frac{2n}{3}) - s$. Finally, let I be the number of marbles that Phase 1 examined but did not swap. We clearly have:

$$S = S_1 + 2S_2 + S_3$$

$$R = 2S_1 + 3S_2 + I$$

Using the notation $x \simeq y$ to mean $x = y + o(n)$ and the relations $E(S_3) \simeq 0$, $E(R) \simeq n$, and $E(I) \simeq \frac{n}{3}$ (derivable in a manner used in the proof of Theorem 2.3) gives

$$E(S) \simeq E(S_1) + 2E(S_2) \qquad (1)$$

$$\frac{2}{3} n \simeq 2E(S_1) + 3E(S_2) \qquad (2)$$

As $n \to \infty$, the fraction of time spent in each state approaches the steady state probability. The probability the chain is in a state which does a single swap is $a + \sum_{i=1}^{smax} c_i + \sum_{i=1}^{smax-1} (d_i + e_i + f_i) = \frac{3\ smax + 3}{5\ smax + 4}$. A double swap is done with probability $b_{smax} = \frac{1}{5\ smax + 4}$. Hence

$$E(S_1) \simeq (3\ smax + 3)\ E(S_2) \qquad (3)$$

Substituting (3) into (2) gives

$$E(S_2) \simeq \frac{2n}{18\ smax + 27}$$

and

$$E(S_1) \approx \frac{6\ \text{smax} + 6}{18\ \text{smax} + 27}\ n$$

Therefore

$$E(S) \approx E(S_1) + 2E(S_2) \approx \frac{6\ \text{smax} + 10}{18\ \text{smax} + 27}\ n. \qquad \square$$

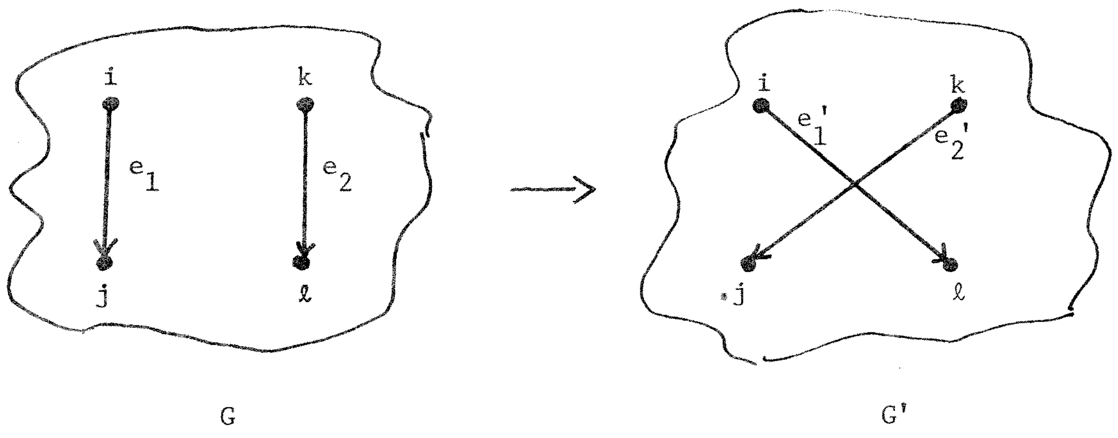

Figure 2.1

The effect of a swap on the corresponding digraphs

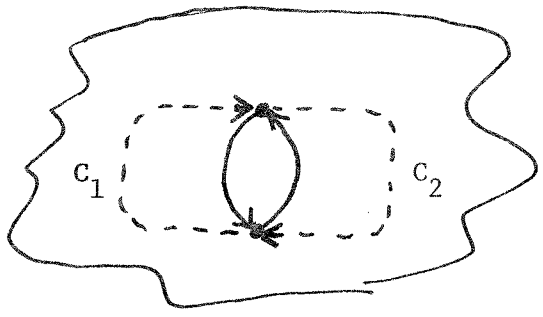

Figure 2.2

The 2-cycle is not in this maximal decomposition. C_1 and C_2 are the two cycles in the decomposition which contain the 2-cycle's edges.

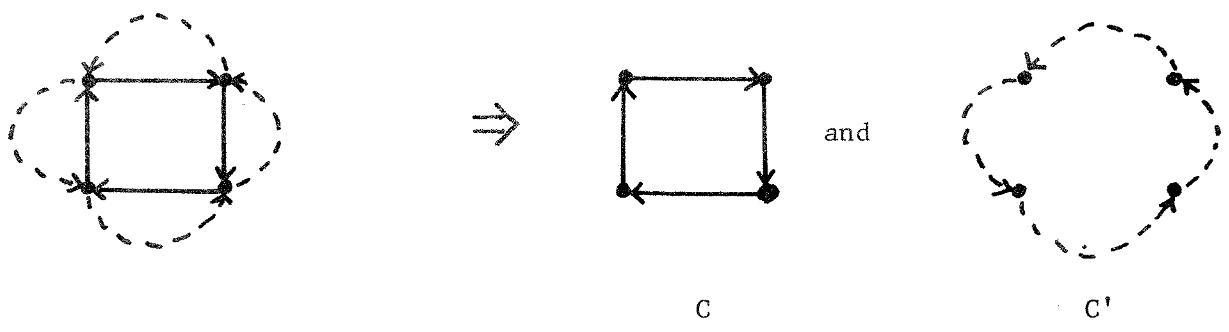

Figure 3.1

A cycle not in the decomposition and the cycles in the decomposition which contain its edges. The situation is not necessarily this simple; the dotted cycles may contain more than one edge of the other cycle and these edges need not be adjacent. However, C' is a cycle in any case.

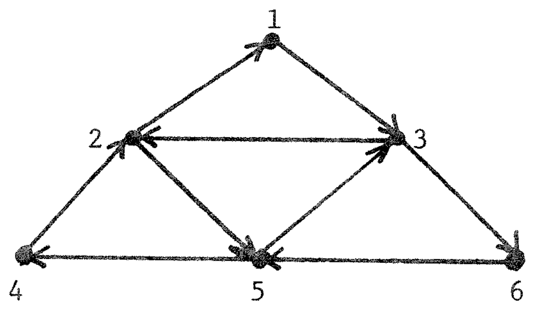

Figure 3.2

A digraph for which the shortest cycle first algorithm might not find a maximal decomposition.

The algorithm might choose cycle 2-5-3 first and obtain a decomposition into two cycles. However, a decomposition into three cycles (2-5-4, 1-3-2, 3-6-5) is possible.

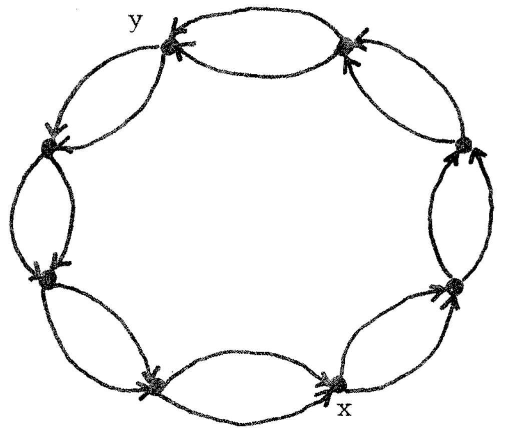

Figure 3.3

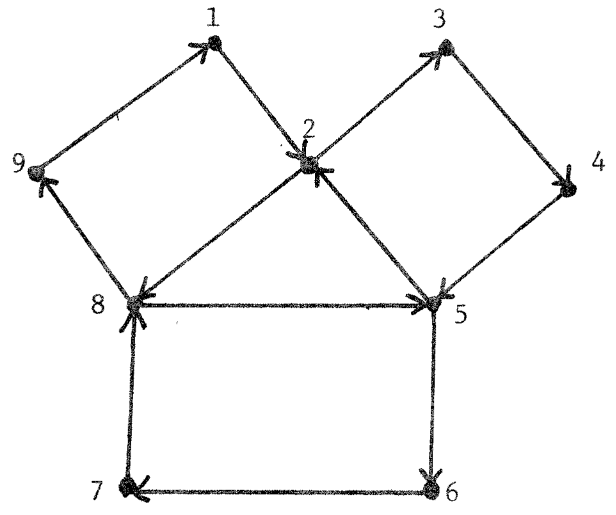

Figure 3.4

A diagraph for which the shortest cycle first algorithm does not find a maximal decomposition.

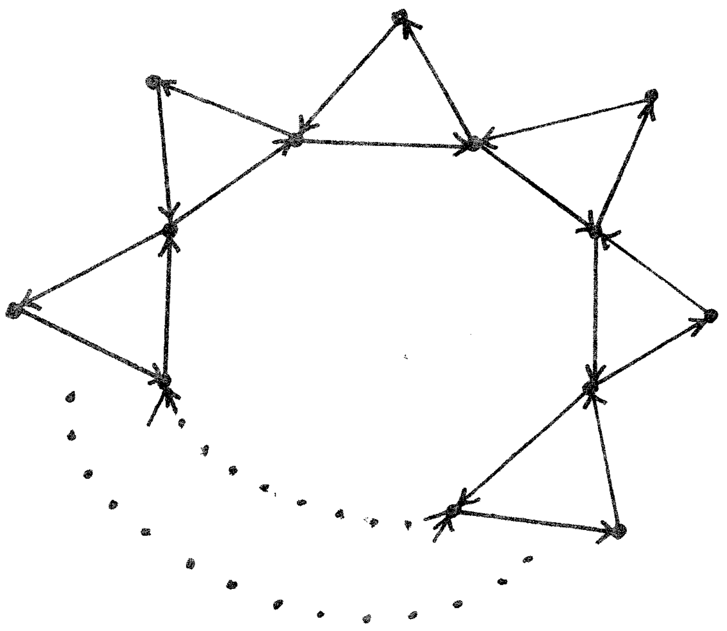

Figure 3.5
A worst case digraph for all 2-cycles first algorithm.

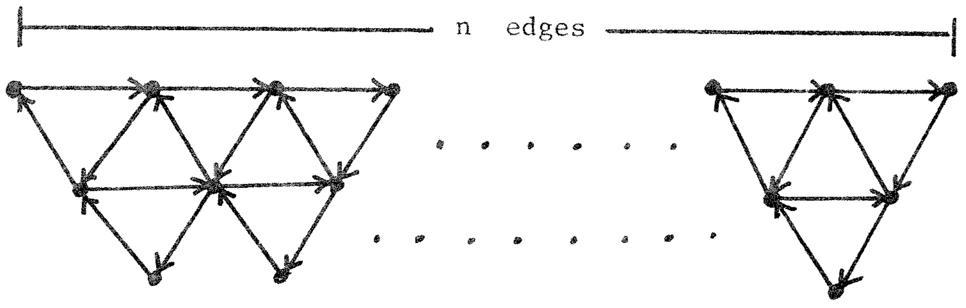

Figure 3.6
The worst case digraph for the shortest cycle first algorithm found so far.

state		conditions
A		ℓ=0 and swap_possible
B_i	$0 \le i \le$ smax	(hascolors(white,blue,red) or hascolors(blue,red,white)) and ℓ=i and (scolor=rcolor or scolor=red)
C_i	$1 \le i \le$ smax	(hascolors(white,blue,red) or hascolors(blue,red,white)) and ℓ=i and (scolor$\neq$rcolor and scolor$\neq$red)
D_i	$1 \le i \le$ smax-1	(hascolors(white,blue,white) or hascolors(blue,blue,white)) and ℓ=i
E_i	$1 \le i \le$ smax-1	(hascolors(white,red,red) or hascolors(blue,red,red)) and ℓ=i
F_i	$1 \le i \le$ smax-1	(hascolors(white,red,white) or hascolors(blue,blue,red)) and ℓ=i

Figure 5.1
The states of the Markov chain

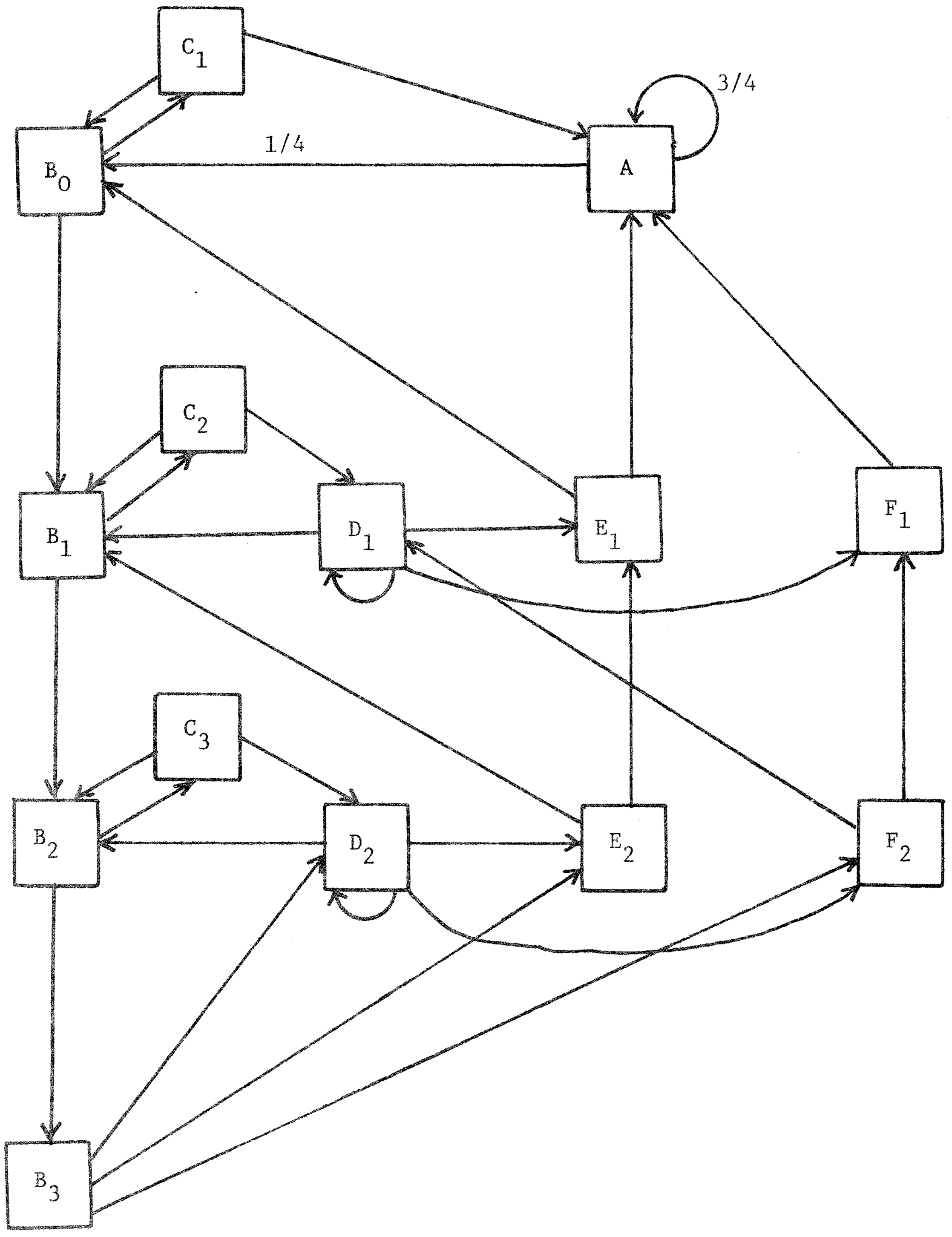

Figure 5.2

The Markov chain used to model the algorithm for smax = 3. The general form should be clear. Except where otherwise indicated, all transitions from the same state are equally likely.

$$
\begin{aligned}
a &= \frac{3}{4}a + \frac{1}{2}c_1 + \frac{1}{2}e_1 + f_1 \\
b_o &= \frac{1}{4}a + \frac{1}{2}c_1 + \frac{1}{2}e_1 \\
b_i &= \frac{1}{2}b_{i-1} + \frac{1}{2}c_{i+1} + \frac{1}{4}d_i + \frac{1}{2}e_{i+1} && ;1 \leq i \leq smax - 2 \\
b_{smax-1} &= \frac{1}{2}b_{smax-2} + \frac{1}{4}b_{smax} + \frac{1}{2}c_{smax} + \frac{1}{4}d_{smax-1} && ;1 \leq i \leq smax \\
b_{smax} &= \frac{1}{2}b_{smax-1} && ;1 \leq i \leq smax - 2 \\
c_i &= \frac{1}{2}b_{i-1} \\
d_i &= \frac{1}{2}c_{i+1} + \frac{1}{4}d_i + \frac{1}{2}f_{i+1} \\
d_{smax-1} &= \frac{1}{4}b_{smax} + \frac{1}{2}c_{smax} + \frac{1}{4}d_{smax-1} \\
e_i &= \frac{1}{4}d_i + \frac{1}{2}e_{i+1} && ;1 \leq i \leq smax - 2 \\
e_{smax-1} &= \frac{1}{4}b_{smax} + \frac{1}{4}d_{smax-1} \\
f_i &= \frac{1}{4}d_i + \frac{1}{2}f_{i+1} && ;1 \leq i \leq smax - 2 \\
f_{smax-1} &= \frac{1}{4}b_{smax} + \frac{1}{4}d_{smax-1}
\end{aligned}
$$

Figure 5.3

The Steady State Equations

References

[1] Dijkstra, E. W., A Discipline of Programming, Prentice-Hall, Englewood Cliffs, N. J., 1976.

[2] Meyer, S. J. A Failure of Structured Programming, Zilog Corp., Software Dept. Technical Report No. 5, Cupertino, Ca., February, 1978.

[3] McMaster, C. L., An Analysis of Algorithms for the Dutch National Flag Problem, CACM 21, 842-846 (Oct. 1978).

[4] Liu C. L., Introduction to Combinatorial Mathematics, McGraw-Hill, New York, 1968, p. 176.

APPENDIX

```
program dnf;
const smax = <number of scouts; smax >= 0.>;
      n = <size of the array to be sorted>;
type  colortype = (red,white,blue);
      indextype = 0..n;
var   w,b : indextype;
      scout : array[0..smax] of indextype;
      scolor,rcolor,wcolor,bcolor : colortype;
      s : 0..smax;
begin
initialize;
while <pointers are valid> do begin
        if swap_possible then single_swap
   else if (s < smax) or (scolor = red) then advance_scout
   else double_swap;
end;
end.

{ We use the following predicates :
   "I" is defined to be
        (( 0 <= i < scout[s] and (0 <= j <= s -> i <> scout[j]) ) ->
                         buck(i) = red) and
        ((n div 3 <= i < w) -> buck(i) = white) and
        ((2*n div 3 <= i < b) -> buck(i) = blue) and
        rcolor <> red and wcolor <> white and bcolor <> blue and
        ((0 < i < s) -> buck(scout[i]) = rcolor) and
        0 <= s <= smax.

   "J" is defined to be
        rcolor = buck(scout[0]) and wcolor = buck(w) and
        bcolor = buck(b) and scolor = buck(scout[s])

        "I and J" is the loop invariant of the main program.   }

procedure initialize;
begin
s := 0;
scout[0] := 0;           advance(scout[0],red,rcolor);
w := n div 3 - 1;        advance(w,white,wcolor);
b := 2*n div 3 - 1;      advance(b,blue,bcolor);
end;


function swap_possible : boolean;
begin
swap_possible := has_pair(rcolor,wcolor,bcolor) or
                 has_pair(scolor,wcolor,bcolor);
end;


function has_pair(r,w,b : colortype) : boolean;
begin
has_pair := ((r = white) and (w = red)) or
            ((r = blue) and (b = red))  or
            ((w = blue) and (b = white))
end;
```

```
{ I and J and swap_possible }
procedure single_swap;
var t : 0..smax;
    tcolor : colortype;
begin
if scolor = red then begin
        t := 0;
        tcolor := rcolor;
end
else begin
        t := s;
        tcolor := scolor;
end;
if (tcolor = white) and (wcolor = red) then begin
        swap(scout[t],w);
        advance_r;
        advance(w,white,wcolor);
end
else if (tcolor = blue) and (bcolor = red) then begin
        swap(scout[t],b);
        advance_r;
        advance(b,blue,bcolor);
end
else { (wcolor = white) and (bcolor = blue) } begin
        swap(w,b);
        advance(w,white,wcolor);
        advance(b,blue,bcolor);
end;
end;
{ I and J }


{ I and buck(scout[s]) = red and
  ( scolor <> red -> buck(scout[0]) = rcolor ) and
  ( scolor = red  -> buck(scout[0]) = red )          }
procedure advance_r;
var i : 0..smax;
begin
if scolor <> red then scolor := red
   else begin
        if s <> 0 then begin
                for i := 0 to s-1 do scout[i] := scout[i+1];
                s := s-1;
        end;
        if s = 0 then begin
                advance(scout[0],red,rcolor);
                scolor := rcolor;
        end;
   end;
end;
{ I and rcolor = buck(scout[0]) and scolor = buck(scout[s]) }
{ ptr = p0 }
procedure advance(var ptr : indextype; skipcolor : colortype;
                  var finalcolor : colortype);
var x : colortype;
begin
repeat
        ptr := ptr + 1;
        x := buck(ptr);
until x <> skipcolor;
finalcolor := x;
end;
{ ((p0+1 <= i <= ptr-1) -> buck(i) = skipcolor)   and
  (buck(ptr) = finalcolor <> skipcolor)   }


{ not swap_possible and I and J and (s < smax or scolor = red)  }
procedure advance_scout;
begin
if scolor = red then advance(scout[s],red,scolor)
   else begin
        s := s + 1;
        scout[s] := scout[s-1];
        advance(scout[s],red,scolor);
   end;
end;
{ I and J }



{ I and J and not swap_possible and s = smax and scolor <> red }
procedure double_swap;
begin
if scolor = white then swap(scout[smax],b)
                  else swap(scout[smax],w);
swap(w,b);
advance_r;
advance(w,white,wcolor);
advance(b,blue,bcolor);
end;
end;
{ I and J }

.
```