Finite-State Machines

In contrast to the preceding chapter, where we did not place any restrictions on the kinds of programs we were considering, in this chapter we restrict the programs under consideration very severely. We consider only programs whose storage requirements do not grow with their inputs. This rules out dynamically growing arrays or linked structures, as well as recursion beyond a limited depth.

The central piece of mathematics for this chapter is the notion of an "equivalence relation." It will allow us to arrive at a complete characterization of what these restricted programs can do. Applying mathematical induction will provide us with a second complete characterization.

State Transition Diagrams

The program `oddlength`, once more, in a picture

Consider the program `oddlength` from Figure 1.1 on page 8. Let's use a debugger on it and set a breakpoint just before the `getchar()` statement. When the program stops for the first time at that breakpoint, we can examine the values of its lone variable. The value of `toggle` is `FALSE` at that time and the "instruction pointer" is of course just in front of that `getchar()` statement. Let's refer to this situation as state A.

If we resume execution and provide an input character, then at the next break the value of `toggle` will be `TRUE` and the instruction pointer will of course again be just in front of that `getchar()`. This is another "state" the program can be in. Let's call it state B.

Let's give the program another input character. The next time our debugger lets us look at things the program is back in state A. As long as we keep feeding it input characters, it bounces back and forth between state A and state B.

What does the program do when an `EOF` happens to come around? From state A the program will output `No`, from state B `Yes`.

Figure 2.1 summarizes all of this, in a picture. The double circle means that an `EOF` produces a `Yes`, the single circle, a `No`. Σ is the set of all possible input

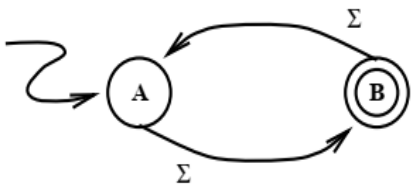

Figure 2.1: The program `oddlength` as a finite-state machine

characters, the *input alphabet*. Labeling a transition with Σ means that any input character will cause that transition to be made.

Terminology A picture such as the one in Figure 2.1 describes a *finite-state machine* or a *finite-state automaton*. States drawn as double circles are *accepting states*, single circles are *rejecting states*. Input characters cause *transitions* between states. Arrows that go from state to state and are labeled with input characters indicate which input characters cause what transitions. For every state and every input character there must be an arrow starting at that state and labeled by the input character. The initial state, or *start state*, is indicated by an unlabeled arrow.[1]

An input string that makes the machine end up in an accepting (rejecting) state is said to be *accepted* (*rejected*) by the machine. If M is a finite-state machine M then $L_M = \{x : x \text{ is accepted by } M\}$ is the *language accepted by* M.

Another Example Let L_{MATCH} be the language of all binary strings that start and end with a 0 or start and end with a 1. Thus $L_{\text{MATCH}} = \{0, 1, 00, 11, 000, 010, 101, 111, 0000, 0010, \ldots\}$ or, more formally,

$$L_{\text{MATCH}} = \{c, cxc : x \in \{0,1\}^\star, c \in \{0,1\}\}. \tag{2.1}$$

Can we construct a finite-state machine for this language? To have a chance to make the right decision between accepting and rejecting a string when its last character has been read, it is necessary to remember what the first character and the last character were. How can a finite-state machine "remember" or "know" this information? The only thing a finite-state machine "knows" at any one time is which state it is in. We need four states for the four combinations of values for those two characters (assuming, again, that we are dealing only with inputs consisting entirely of 0s and 1s). None of those four states would be appropriate for the machine to be in at the start, when no characters have been read yet. A fifth state will serve as start state.

[1] Purists would call the picture the "transition diagram" of the machine, and define the machine itself to be the mathematical structure that the picture represents. This structure consists of a set of states, a function from states and input symbols to states, a start state, and a set of accepting states.

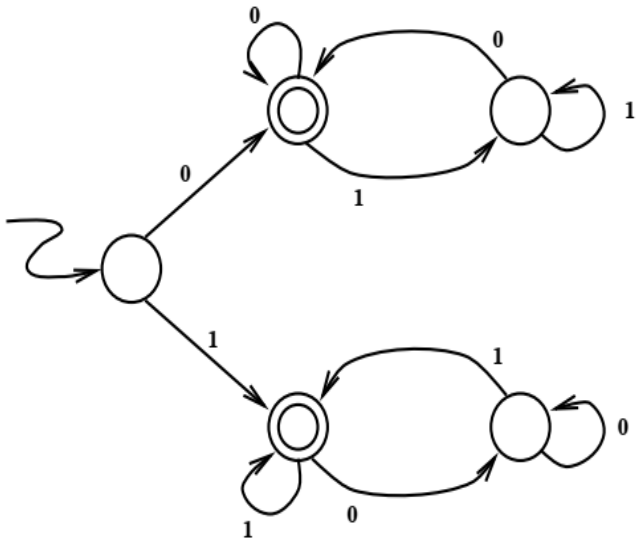

Figure 2.2: A finite-state machine that accepts L_{MATCH}

Now that we know what each state means, putting in the right transitions is straightforward. Figure 2.2 shows the finished product.

(Note that we could have arrived at the same machine by starting with the program `match` in Figure 2.3 and translating it into a finite-state machine by the same process that we used earlier with the program `oddlength`.)

A third example

Consider the language $L_{\text{EQ-4}}$ of all nonempty bitstrings of length 4 or less which contain the same number of 0s as 1s. Thus $L_{\text{EQ-4}} = \{01, 10, 0011, 0101, 0110, 1001, 1010, 1100\}$.

What does a finite-state machine need to remember about its input to handle this language? We could make it remember exactly what the input was, up to length 4. This results in a finite-state machine with the structure of a binary tree of depth 4. See Figure 2.4. Such a binary tree structure corresponds to a "table lookup" in a more conventional language. It works for any finite language.

The preceding example proves

Theorem 10 *Every finite language is accepted by a finite-state machine.*

Minimizing Finite-State Machines

The finite-state machine of Figure 2.4 is not as small as it could be. For example, if we "merged" the states H, P, Q, and FF — just draw a line surrounding them and regard the enclosed area as a new single state —, we would have reduced the size of this machine by three states (without changing the language it accepts).

Why was this merger possible? Because from any of these four states, only rejecting states can be reached. Being in any one of these four states, no matter what the rest of the input, the outcome of the computation is always the same, *viz.* rejecting the input string.

```
void main( void )
{
    if (match())
        cout << "Yes";
    else
        cout << "No";
}

boolean match( void )
{
    char first, next, last;

    first = last = next = getchar();
    while (next != EOF)
    {
       last = next;
       next = getchar();
    }
    return ((first == last) && (first != EOF)):
}
```

Figure 2.3: Another "finite-state" program, `match`

Can J and K be merged? No, because a subsequent input 1 leads to an accepting state from J but a rejecting state from K.

What about A and D? They both have only rejecting states as their immediate successors. This is not a sufficient reason to allow a merger, though. What if the subsequent input is 01? Out of A the string 01 drives the machine into state E — a fact usually written as $\delta(A, 01) = E$ — and out of D the string 01 drives the machine into state $\delta(D, 01) = Q$. One of E and Q is rejecting, the other accepting. This prevents us from merging A and D.

When is a merger of two states p and q feasible? *When it is true that for all input strings s, the two states $\delta(p, s)$ and $\delta(q, s)$ are either both accepting or both rejecting.* (Note that s could be the empty string, which prevents a merger of an accepting with a rejecting state.) This condition can be rephrased to talk about when a merger is *not* feasible, *viz. if there is an input string s such that of the two states $\delta(p, s)$ and $\delta(q, s)$ one is rejecting and one is accepting.* Let's call such a string s "a reason why p and q cannot be merged." For example, the string 011 is a reason why states B and C in Figure 2.4 cannot be merged. The shortest reasons for this are the strings 0 and 1. A reason why states D and E cannot be merged is the string 11. The shortest reason for this is the empty string.

Figure 2.5 is an illustration for the following observation, which is the basis for an algorithm for finding all the states that can be merged.

Observation 3 *If c is a character and s a string, and cs is a shortest reason why p and q cannot be merged, then s is a shortest reason why $\delta(p, c)$ and $\delta(q, c)$*

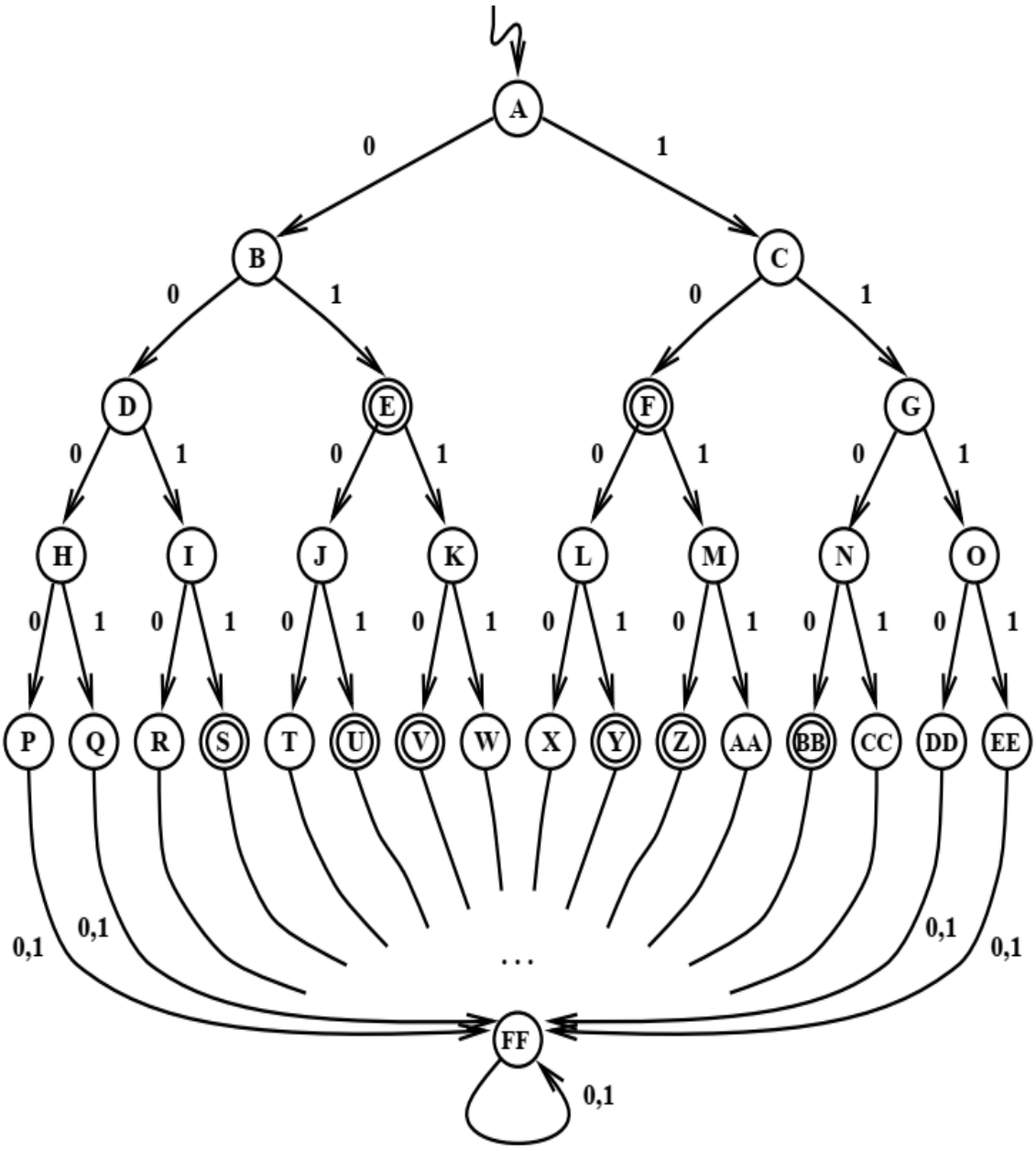

Figure 2.4: A finite-state machine for $L_{\text{EQ-4}}$

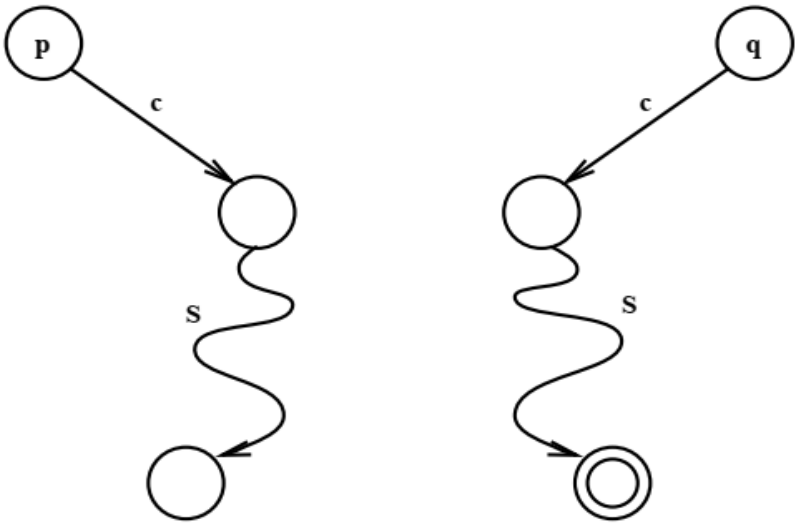

Figure 2.5: Illustrating Observation 3

cannot be merged.

Observation 3 implies that the algorithm is Figure 2.6 is correct.

When carrying out this algorithms by hand, it is convenient to keep track of clusters of states.[2] Two states p and q are in the same cluster if $(p, q) \in$ `MergablePairs`. As we find reasons for not merging states, i.e., as we find the condition of the `if`-statement to be true, we break up the clusters. If p and q are two states that cannot be merged and the shortest reason for that is a string of length $k > 0$, then p and q will remain together through the first $k-1$ passes through the `repeat`-loop and get separated in the k^{th} pass.

For example, consider the finite-state machine in Figure 2.7. It accepts "identifiers with one subscript" such as `A[2]`, `TEMP[I]`, `LINE1[W3]`, `X034[1001]`. Figure 2.8 shows how the clusters of states develop during execution of the minimization algorithm. States a and b get separated in the third pass, which is consistent with the fact that the shortest strings that are reasons not to merge a and b are of length 3, for example "`[0]`."

Other examples are the finite-state machines in Figures 2.9 and 2.4. On the finite-state machine of Figure 2.9 the algorithm ends after one pass through the loop, resulting in the machine shown in Figure 2.10. The binary-tree-shaped machine of Figure 2.4 takes three passes, resulting in the machine shown in Figure 2.11.

[2]Note: This works because the binary relation MergablePairs is always an equivalence relation.

```
MergablePairs = ( States × States )
                − ( AcceptingStates × RejectingStates )
repeat
    OldMergablePairs = MergablePairs
        for all (p, q) ∈ MergablePairs
            for all input characters c
                if ( (δ(p, c), δ(q, c)) ∉ OldMergablePairs )
                    MergablePairs = MergablePairs − (p, q)
until
    MergablePairs = OldMergablePairs
```

Figure 2.6: An algorithm for finding all mergable states in a finite-state machine

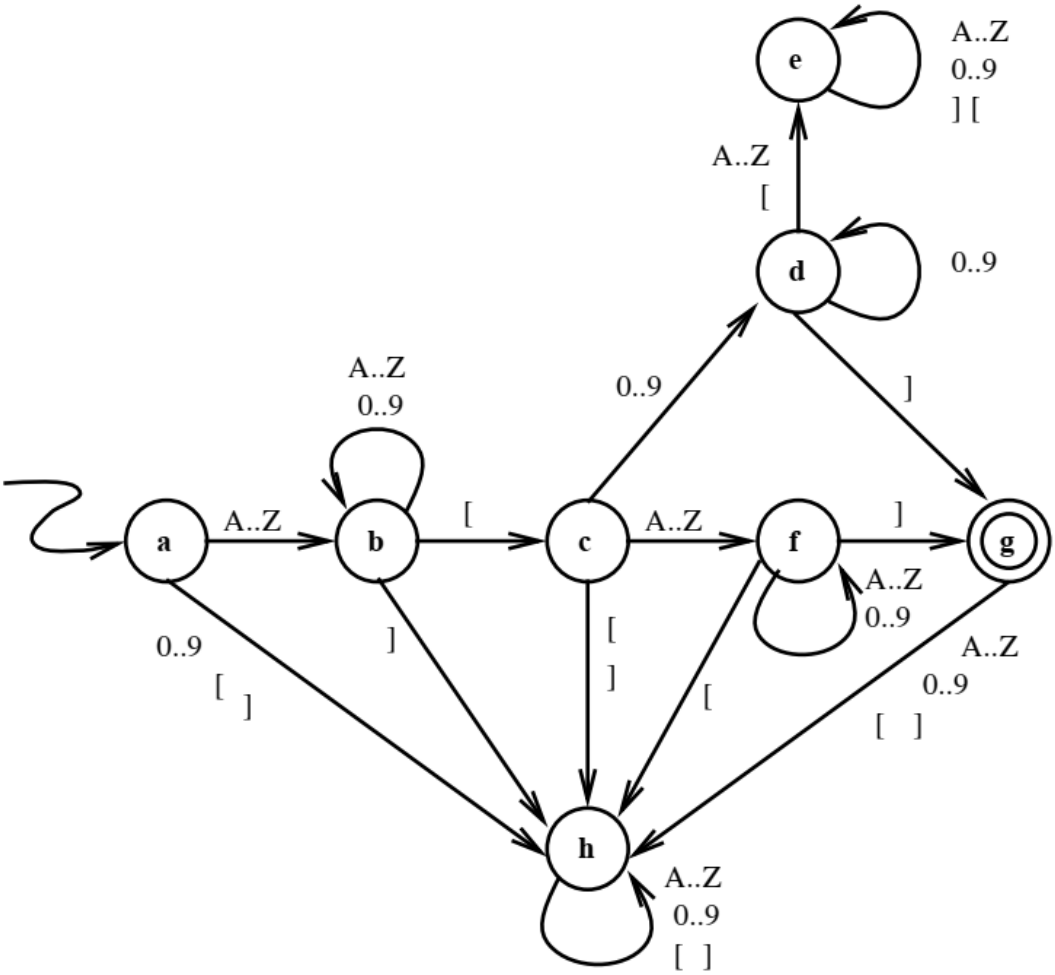

Figure 2.7: A finite-state machine for accepting identifiers with one subscript

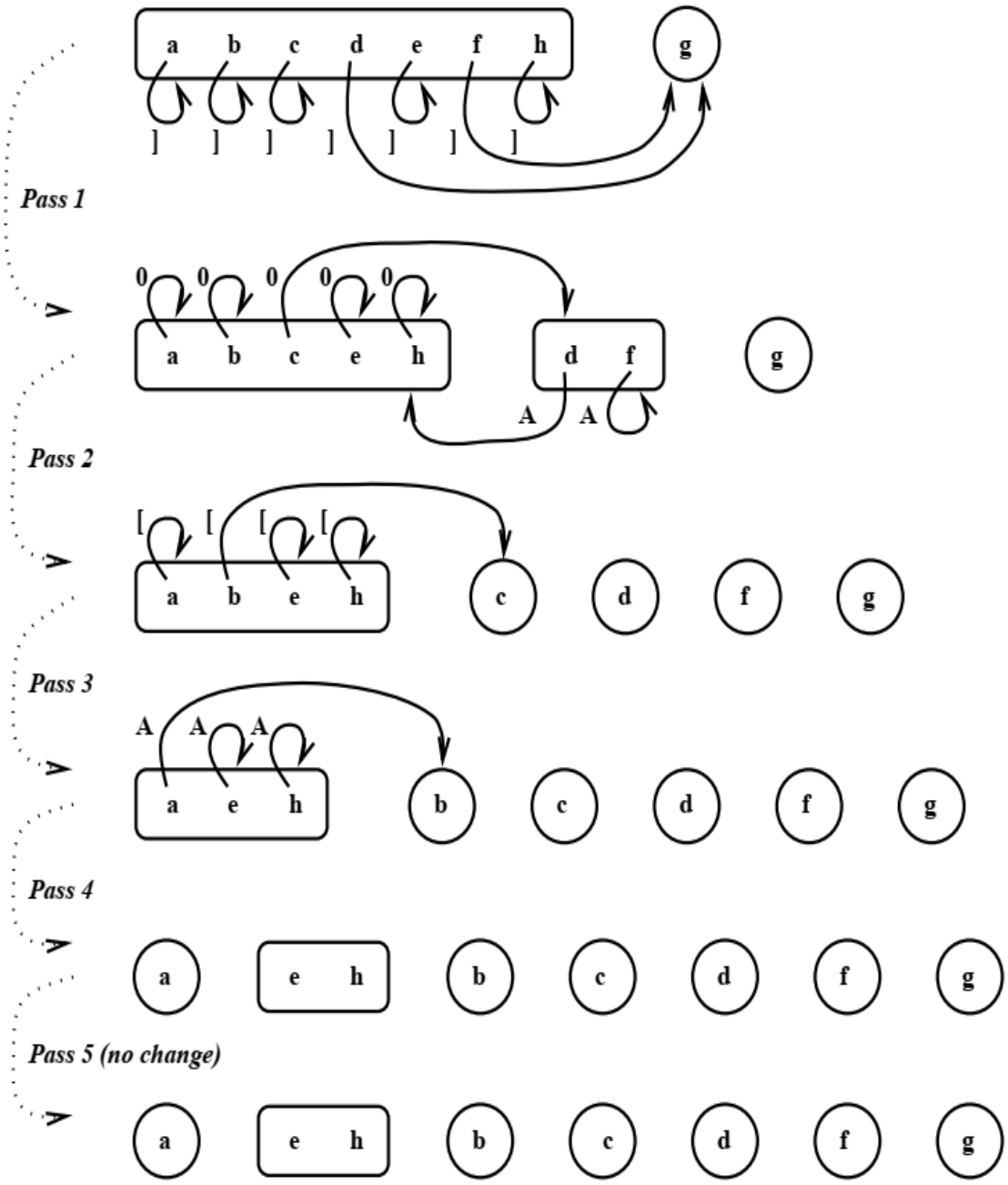

Figure 2.8: Minimizing the finite-state machine of Figure 2.7

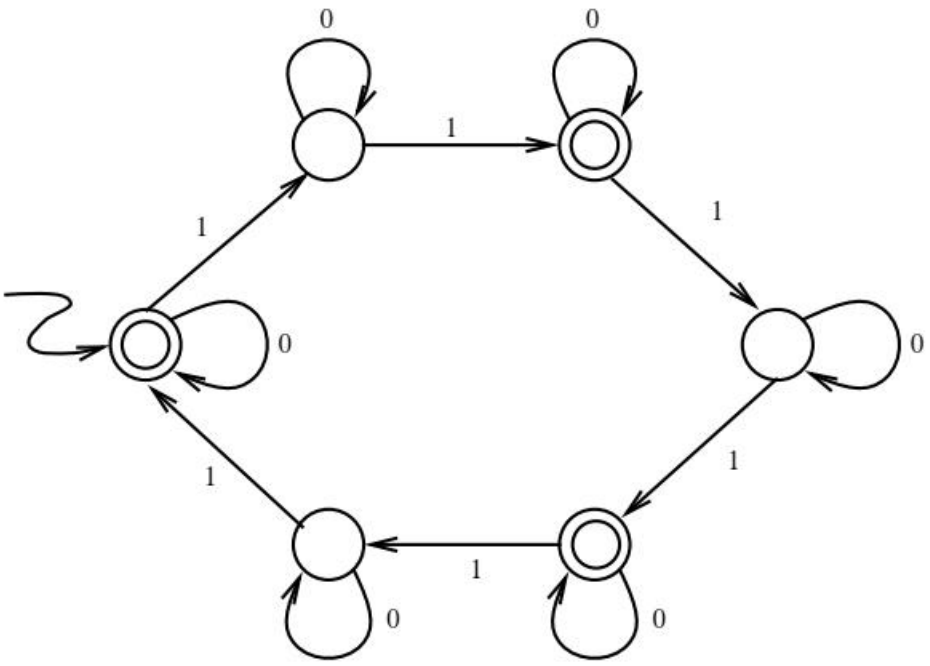

Figure 2.9: A finite-state machine for checking parity

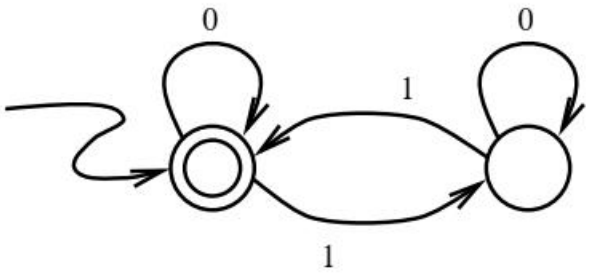

Figure 2.10: A minimal finite-state machine for checking parity

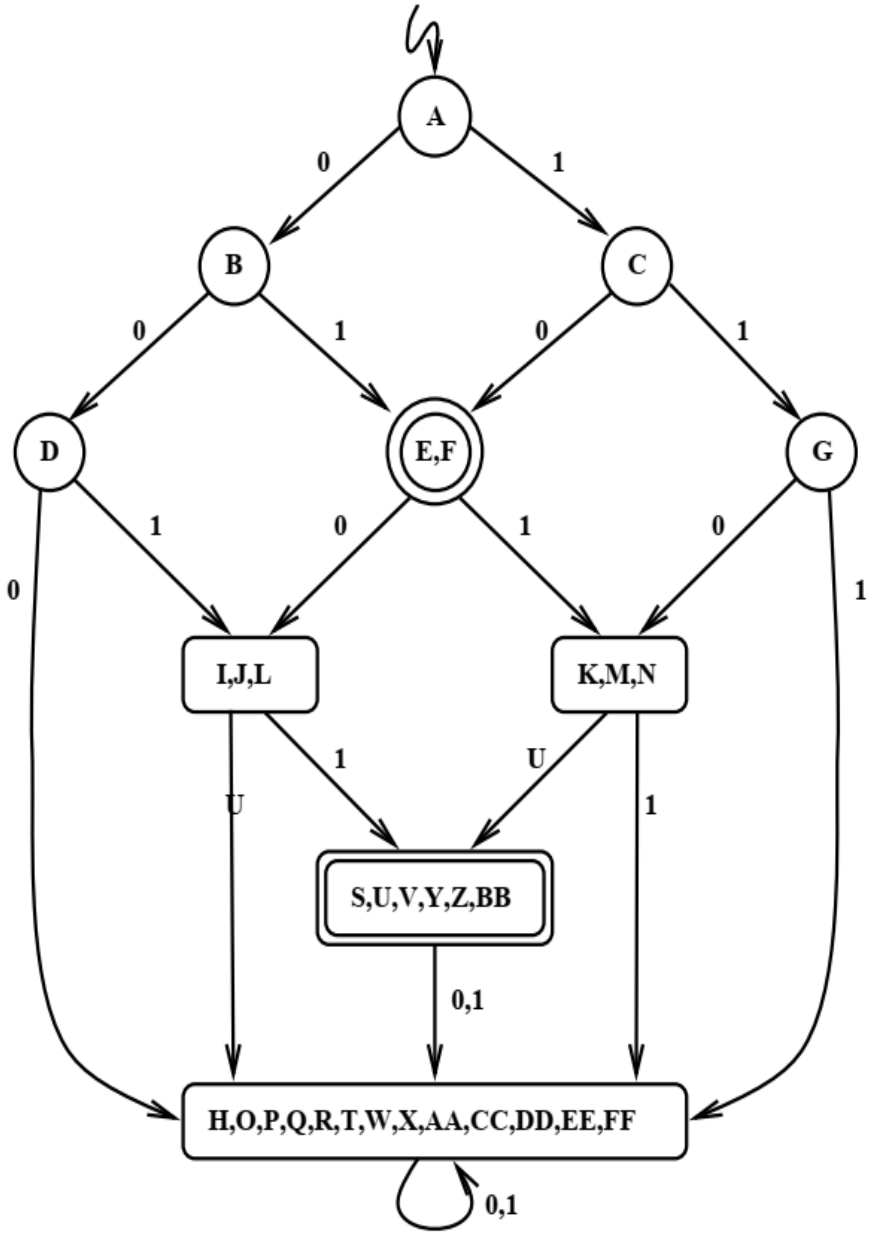

Figure 2.11: A minimal finite-state machine for $L_{\text{EQ-4}}$

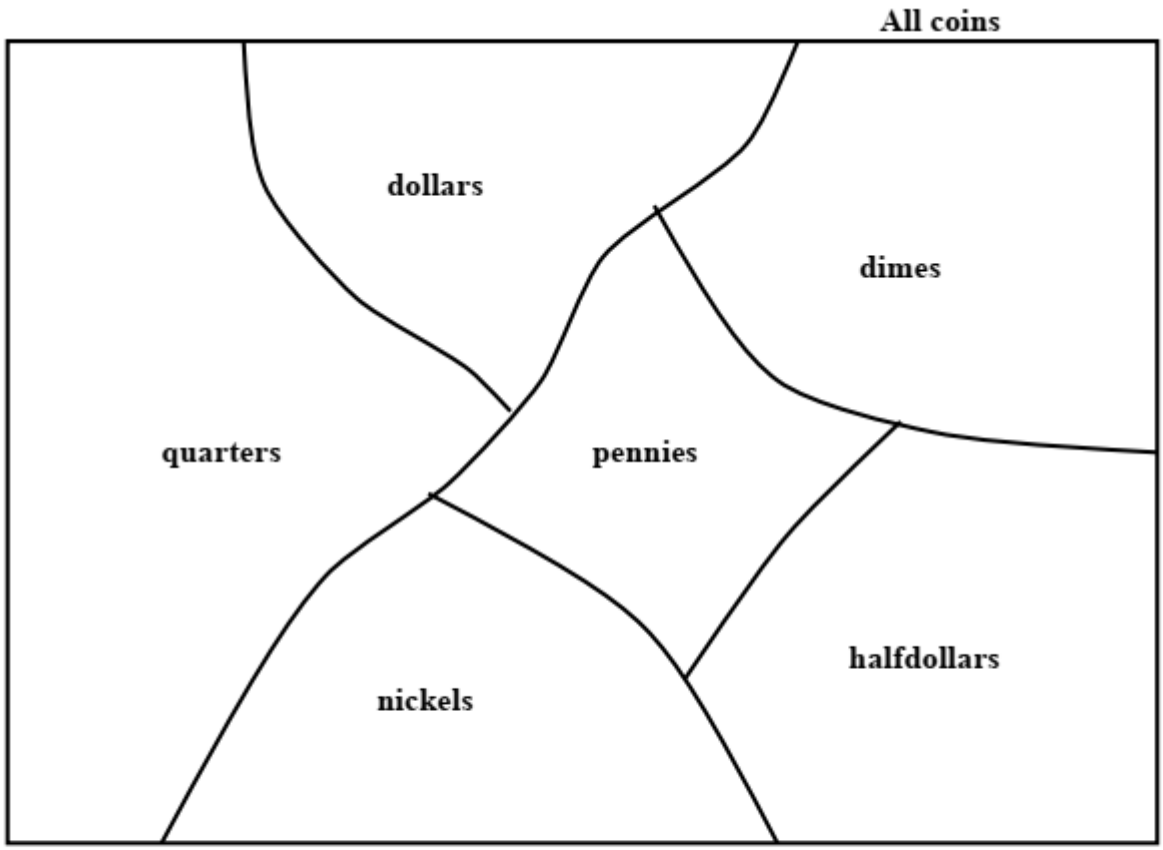

Figure 2.12: Equivalence classes of coins