Journal of Graph Algorithms and Applications
http://www.cs.brown.edu/publications/jgaa/
vol. 6, no. 2, pp. 157–173 (2002)

Traversing Directed Eulerian Mazes

S. Bhatt
Akamai Technologies
500 Tech Square, Cambridge MA 02139
bhatt@akamai.com

S. Even
Computer Sci. Dept., Technion – Israel Inst. of Tech.
Haifa, Israel
even@cs.technion.ac.il

D. Greenberg
Center for Computing Science
Bowie, MD
dsg@super.org

R. Tayar
Computer Sci. Dept., Technion – Israel Inst. of Tech.
Haifa, Israel
csrafi@cs.technion.ac.il

Abstract

The paper describes two algorithms for threading unknown, finite directed Eulerian mazes. Each of these algorithms is performed by a traveling robot whose control is a finite-state automaton. It is assumed that each vertex has a circular list of its outgoing edges. The items of this list are called exits. Each of the algorithms puts in one of the exits of each vertex a scan pebble. These pebbles can be used by a simple robot as traffic signals, which allow it to traverse an Eulerian cycle of the maze.

For a directed graph (maze) $G(V, E)$, the simple algorithm performs $O(|V| \cdot |E|)$ edge traversals, while the advanced algorithm traverses every edge three times. Let $d_{out}(v)$ be the out-degree of vertex v. The algorithms use, at each vertex v, a local memory of size $O(\log d_{out}(v))$.

Communicated by S. Khuller: submitted January 2002; revised June 2002

Work by S. Even supported by the Fund for the Promotion of Research at the Technion.

1 Introduction and Model

The question of how best to traverse an unknown maze, given only local knowledge, has been long studied. [8, 3, 5, 7, 6, 1, 4, 2]. Without the ability to store some knowledge about the maze the searcher can easily become trapped in cycles.

In this paper we consider the case of directed Eulerian mazes/graphs, and show that a finite state automaton, with access to some local memory stored in the vertices, can find an Eulerian cycle. During the search for this cycle, each edge is traversed three times.

It is assumed that each vertex has a circular list of its outgoing edges. The items of this list are called *exits*.

The challenge is to use only as simple a "robot," R, as possible. R has a finite automaton control and a few external actions. The external actions affect the external state which consists of the following:

- The vertex in which R is at, and more specifically, the exit at which R is at.
- For every vertex v, the location of some small constant number of pebbles at exits of v, and the value of a constant number of flags' bits at v.

Let $d_{out}(v)$ be the out-degree of v. The external actions allow R to access $O(\log d_{out}(v))$ bits of information, stored at v. Note that R's memory may not be sufficient to store all these bits at any one time, since no bound on $d_{out}(v)$ is assumed.

The robot R can perform two essential movement actions:

- R can traverse an edge from its beginning, which is an exit of the vertex it is at, to its end, to enter another (the same, in case of a self-loop) vertex.
- R can move from the exit it is at to the next/previous exit in the cyclic order of the exits of the vertex it is at.

Also, R can perform two pebbling actions:

- R can observe if a pebble of a specified type is located at the exit R is at.
- R can put/remove any pebble at/from the exit it is at.

Finally, R can perform two flagging actions:

- R can observe whether a certain flag is turned on at the vertex R is at.
- R can turn on/off a certain flag at the vertex R is at.

For simplicity we will often use compound actions such as *traverse the edge with the scan pebble* which are straightforward finite automata functions of the basic actions.

The problem considered in this paper can now be restated as follows: Design a robot R so that when it is placed at any vertex of any directed Eulerian finite

graph, with any cyclic ordering of the exits of each vertex, R will eventually halt, leaving a scan pebble at one exit of each vertex. These scan pebbles serve as traffic signals for a simple scanning robot. Using these pebbles, and moving them as it goes along, this robot will traverse an Eulerian cycle, will halt, and when it does, the scan pebbles will be exactly where they have been when the simple robot started this traversal.

2 History of Maze Traversal

In 1967, Michael Rabin [8] considered the problem of threading undirected connected graphs by means of a finite automaton which has some fixed number of pebbles it can leave at vertices and which it can move from one vertex to another. Note that by threading we mean visiting every vertex, at least once. He proved that no such automaton can thread all finite undirected graphs. It follows that no finite automaton can thread all finite, connected graphs, unless some unbounded information can be stored in the vertices.

In view of this impossibility result, it is no wonder that the classical (19-th century) algorithms for threading undirected mazes, such as that of Trémaux's and Tarry's (see, for example Even's book [5], Chapter 3) store data in the vertices. More about threading undirected mazes can be found in the book of Hemmerling [7]. However, these algorithms are not easy to implement on directed graphs due to the fact that backtracking along edges is no longer allowed.

Notice that every connected undirected graph can be transformed to a directed Eulerian graph by replacing every edge u—v with two directed edges $u \longrightarrow v$ and $v \longrightarrow u$. Thus, the impossibility result for threading undirected graphs implies the impossibility of threading for directed Eulerian graphs. This, in turn, proves the impossibility of finding an Euler cycle in Eulerian directed graphs.

Therefore, in order to thread all finite directed Eulerian graphs, it is natural to allow our robot, R, to store and use some information in the vertices.

Even, Litman and Winkler [6] presented an algorithm of time complexity $O(|V|^2)$ to thread directed networks. However, their algorithm assumes a different computational model — each vertex is a finite automaton and the directed edges are communication links. Afek and Gafni [1] present a fairly complicated solution which uses essentially Depth-First Search (like Trémaux's algorithm), and includes a method to effectively backtrack on directed edges. Their method uses $O(|V| \cdot |E| + |V|^2 \cdot \log |V|)$ edge traversals, and requires that vertices are labeled with distinct names. We shall later say more about the comparison of our results with those of Afek and Gafni.

Deng and Papadimitriou [4] also use a more powerful model than we do. Their purpose is to explore the graph; i.e. to discover its structure. They assume that vertices have distinct identities and that the robot is a general purpose computer (Turing equivalent). They describe a recursive algorithm which finds an Eulerian cycle in a directed Eulerian graph. This algorithm traverses each edge twice. They build on this algorithm to attack the problem

of exploring general strongly connected directed graphs. (See, also, Albers and Henzinger [2].)

3 Overview of the Robot Algorithms

Let $G(V, E)$ be a finite Eulerian directed graph. Namely, G's underlying undirected graph is connected, and for every vertex $v \in V$, $d_{in}(v) = d_{out}(v)$, i.e., its indegree is equal to its outdegree. Euler's Theorem implies that G has a directed cycle in which every edge of G appears exactly once.

Since $d_{in}(v) = d_{out}(v)$ we shall simply use $d(v)$ to denote $d_{out}(v)$.

3.1 The Simple Algorithm

Without any pebbles the robot, R, might be doomed to pick the same exit every time it visited a vertex. In this algorithm, R uses a scan pebble to "remember" which is the next exit to be taken. At each vertex, R moves to the exit with the pebble, call it E_p, places the pebble at the next exit, returns to E_p, and traverses that edge. As we will show below, the repetition of this simple "move pebble forward before traverse" alone will eventually establishes an Eulerian tour, i.e., after wandering for a while, R will traverse an Eulerian cycle, and repeat it, indefinitely, unless some mechanism causes it to halt.

Afek and Gafni [1] used a similar idea in their Traversal-1 algorithm. We use a different halting mechanism (employing a two bits flag at each vertex and using a single bit of internal state of R,) and take advantage of the fact that our graphs are assumed to be Eulerian, while they assumed only strong connectivity. We will show that this Simple algorithm makes $O(|V| \cdot |E|)$ edge traversals before it halts, using our halting rule.

Afek and Gafni [1] show that their scanning method is existentially optimal, that is, that there exists a family of (non-Eulerian) directed graphs which require $\Omega(|V| \cdot |E|)$ edge traversals. However, as is shown in the Appendix, there are *dense* (Eulerian) graphs for which any Depth-First Search approach, including that of Afek and Gafni, takes $\Omega(|V| \cdot |E|) = \Omega(|V|^3)$ edge traversals, while our variant takes $O(|V|^2)$.

3.2 The Advanced Algorithm

To prepare for our second robot algorithm we first describe an algorithm which does not meet our traveling robot model. We first prove the validity of this algorithm and later show that we can return to our model and preserve the validity.

The algorithm consists of two phases. The first phase, called *preparatory*, is nondeterministic. It concludes by placing a scan pebble at one exit of each vertex, thus, in fact, having found an Eulerian tour. The second phase, called *stroll*, can then execute the simple traveling robot algorithm to trace an Eulerian

cycle. When it halts the pebbles are back where the preparatory phase had placed them.

Finally, our Advanced Algorithm replaces the nondeterministic preparatory phase with a traveling robot sub-algorithm. The new preparatory phase requires two more pebbles per vertex and two additional 3-value variables in each vertex. Including both its phases, the Advanced Algorithm traverses each edge 3 times.

3.3 The Recursive Algorithm

It is worth noting that the Recursive Algorithm of Deng and Papadimitriou can also be performed by a traveling finite robot. However, the required memory in each vertex v is $\Theta(d(v) \cdot \log d(v))$. If the robot is "input aware", (i.e. when a vertex is entered the output edge is a constant 1-1 function of the input edge,) then the Recursive algorithm need scan every edge only twice. Without input awareness, each edge is scanned exactly 3 times. This was shown by Tayar [9].

Finally, we note that while each of the three algorithms (Simple, Advanced and Recursive) produces an Eulerian cycle, they may produce different cycles (e.g. when applied to the maze shown in Figure 1.)

4 The Simple Algorithm

4.1 The Algorithm

The traversal of the graph is to be performed by a finite-state robot R. The algorithm which governs the behavior of R consists of two mechanisms: The *scanning mechanism*, which determines the order in which the edges are traversed, and the *halting mechanism*.

4.1.1 The Scanning Mechanism

In each vertex v there is a *scan* pebble, initially placed at some arbitrary exit. Each time the robot enters a vertex it finds the scan pebble, moves it forward one position, and exits through the edge where the scan pebble has been.

Clearly, the scanning continues indefinitely, unless the halting mechanism stops it.

4.1.2 The Halting Mechanism

For the purpose of deciding when R stops scanning, vertices and exits are marked as follows:

- Each vertex has a root flag r. Initially $r = 0$ at all vertices. The first action by the robot is to mark its initial vertex flag $r = 1$. Thereafter, no vertex has its r flag changed.
- Each vertex also has a status flag, s, which is initially marked NEW. On first visiting a vertex the robot changes s to SEEN. Eventually, the

robot will change each s flag to DONE and thereafter leave it unchanged. Also, on the first visit the robot finds the exit in which the scan pebble is located and puts the *start* pebble at the same exit. The start pebble is not moved anymore. It helps determine if all exists have been used to leave the vertex.

In addition, the robot carries a state bit F. This bit may be UP or $DOWN$. The initial value of F is $DOWN$. The meaning of $F = UP$ is that on the present "phase", to be defined shortly, all vertices, it has encountered so far, have had all their outgoing edges traversed, at least once.

The actions of the robot under the Simple Algorithm are given in Table 1. Lines 3, 5, 7 and 20 perform the scanning mechanism. Lines 1-2, 4, 8-16 and 18-19 perform the halting mechanism. Line 17 defines the end of a phase. The phases are only defined to simplify the proof.

```
1   r ← 1
2   s ← SEEN
3   goto exit with scan pebble
4   put start_edge pebble at exit
5   move scan pebble to next exit and leave vertex through current exit
6   while TRUE
7       goto exit with scan pebble
8       if s = NEW then do
9           put start_edge pebble at exit
10          s ← SEEN
11      if r = 0 then do
12          if s ≠ DONE then do
13              F ← DOWN
14              if start_edge pebble at next exit then
15              s ← DONE
16      if r = 1 and exit has the start_edge pebble then do
17          (end of phase)
18          if F = UP then HALT
19          F ← UP
20      move scan pebble to next exit and leave vertex through current exit
```

Table 1: The Simple Algorithm

4.2 Validity

The vertex where $r = 1$ is called the *root* ρ.

Theorem 1 *The robot R will halt at the root, ρ, after having traversed all edges of G. This will happen within $O(|V| \cdot |E|)$ edge traversals. In its last $|E|$ edge traversals R makes an Eulerian tour.*

Note that upon termination the scan pebbles are not necessarily back in their initial condition. In fact, their position is exactly what has been "learned". If now the traversal is restarted, with $F = UP$, R will perform an Eulerian tour and stop, and the scan pebbles will again be in the "learned" position.

Proof: Note the definition of a phase, as in Table 1, Line 17. First, let us examine the route of R during the first phase. Clearly, every edge incident with ρ has been traversed exactly once. Furthermore, every edge on the route of the first phase is traversed once only. This is easily proved by induction on the order in which the edges are traversed: When a vertex $v \neq r$ is entered by an edge (traversed for the first time) the number of used incoming edges is one greater than the number of used outgoing edges. Thus, the exit in which the pebble is placed leads to an edge which has not yet been traversed.

Note that if during the first phase every incident edge of $v \neq \rho$ has been traversed then v's s flag is $DONE$. Other vertices are still flagged NEW or $SEEN$. However, for every vertex the number of untraversed incoming edges is equal to the number of untraversed outgoing edges.

Now consider the second phase. As long as for every vertex $v \neq \rho$, encountered by R, $s = DONE$, the route of the second phase is identical to that of the first phase. This follows from the fact that, at the end of the first phase, for ρ and every $DONE$ vertex, the scan pebble is back at the start exit. If not all edges of G have been traversed in the first phase, there must be vertices on the first route for which some of their outgoing edges have not been traversed and such vertices are still flagged $SEEN$. This follows from the fact that G is strongly connected. If $SEEN$ vertices exist, let v be the first $SEEN$ vertex encountered in the second phase. A new tour starts at v, using only edges untraversed in the first phase, and ending in v. Now v is flagged $DONE$ and its scan pebble is at the start exit. The route of the first phase is resumed. Since every vertex which is still flagged $SEEN$ and which is on the route of the first phase is similarly treated, at the end of the second phase all vertices encountered on the first route are now flagged $DONE$. Every traversed edge has been traversed exactly once in the second phase, which is the second time for those traversed on the first route.

For every phase $i > 1$, the algorithm maintains the following properties:

- While R is traveling through ρ and $DONE$ vertices, it retraces the route of the $(i-1)'$st phase.
- If R encounters a $SEEN$ vertex, u, it suspends the retracing of the tour of the $(i-1)'$st phase. A new tour starts at u and ends there. u is now $DONE$. The tour of the $(i-1)'$st phase is resumed (using the start exit). Thus, every edge it traversed at most once during a phase.
- If for every vertex $v \neq \rho$, which R encounters, v is $DONE$, then R returns to ρ with $F = UP$, and halts.

During each phase, other than the last and possibly the first, at least one SEEN vertex becomes $DONE$. Thus, there are at most $|V| + 1$ phases. Hence, the number of edge traversals is $O(|V| \cdot |E|)$. □

5 The Generic Algorithm

In order to explain our more efficient algorithm for finding an Eulerian directed cycle in $G(V, E)$, in terms of the number of edge traversals, we describe first a *Generic Algorithm*. This is a nondeterministic algorithm which deviates from our computational model; i.e. it is not limited to actions performed by a traveling finite automaton. Later on we show how to remove the nondeterminism and run the algorithm by means of a finite automaton which threads the directed graph.

5.1 Definitions

- An edge is *new* if it has not been traversed yet. It is *old* otherwise.
- A vertex is *new* if all its outgoing edges are new. It is *old* otherwise.
- An **exploration** from vertex v is a directed path which starts in v and continues via a new directed edge, as long as there is one. It is assumed that, in each vertex, exits are chosen according to the specified circular order.

For convenience, we shall assume that the exits of a vertex v are labeled $v.1, v.2, \ldots, v.d$, where $v.1$ is the start exit, as we used it in Section 4, d is $d_{out}(v)$ and the exits are numbered according to the specified circular order. Also, we shall use a scan pebble, one per vertex, to be placed at one of the exists.

5.2 The Preparatory Algorithm

Consider Preparatory, as stated in Table 2.

```
choose a vertex ρ
run an exploration from ρ
while there are old vertices which have new exits do
      choose such a vertex u
      run an exploration from u
for every vertex v do
      if only one exploration passed through v then
            put v's pebble at the first exit of this exploration
      else  put the pebble at the first exit of the
            second exploration which passed through v
```

Table 2: Preparatory

Note that the first exploration ends at ρ, the vertex at which it started. This observation, like the proof of Euler's theorem, follows from the fact that

whenever any vertex u, other than ρ, is entered, the number of old incoming edges is greater (by 1) than the number of old outgoing edges, and since $d_{in}(u) = d_{out}(u)$, there is at least one new outgoing edge. When the first exploration ends, for every vertex the number of new incoming and outgoing edges is the same, and this invariant holds every time an exploration ends.

For every vertex, Preparatory places a pebble at one of its exits. The position of these pebbles is the information *learned* by running Preparatory.

5.3 The Scan Algorithm

The benefit of the Preparatory Algorithm becomes evident when it is followed by the Scan Algorithm. The Scan Algorithm is similar to the actions taken by R in the last phase of the Simple Algorithm. The pebble is used just as the scan pebble has been used there. However, the locations of the pebbles, when Scan begins, may be different from those of the scan pebbles, when the last phase of Simple starts. Scan halts when exit $\rho.1$ is about to be used again. The route R takes during Scan is an Eulerian tour.

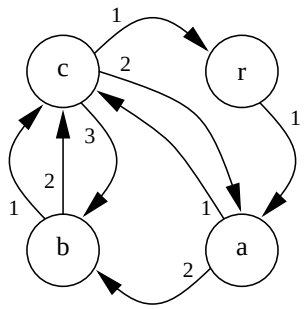

Figure 1: Example for running Generic

The interaction of Preparatory with Scan may seem a bit magical so we demonstrate their action on the example shown in Figure 1. For each vertex, the exits are numbered, and for convenience, all vertices are given names. Assuming $\rho = r$, the first exploration yields the cycle $C_1 : r.1, a.1, c.1$. Assume the second exploration starts at a; it yields a cycle $C_2 : a.2, b.1, c.2$. Assume the third cycle is $C_3 : b.2, c.3$. Since only one exploration cycle includes r, a pebble is placed on $r.1$. The other vertices are in more than one exploration so the pebbles are placed at the first edge used by the second exploration, that is, $a.2, b.2$ and $c.2$. This completes the operation of Preparatory.

Now, during a Scan, the following Eulerian cycle is traversed:

$$r1, a2, b2, c2, a1, c3, b1, c1.$$

Note that the edges of a Preparatory cycle may not be scanned consecutively, and they may even appear in an order different from the order in which they have been traversed in the exploration. Nonetheless, as we show below, Scan will always traverse an Eulerian path.

Lemma 1 *In the directed path indicated by Scan, no edge is traversed more than once.*

Proof: The only exploration which passes through ρ is the first one, and therefore Preparatory places the pebble at exit $\rho.1$. Scan halts when R is at ρ and the pebble is back at exit $\rho.1$ when ρ is entered. Thus, every edge out of ρ has been traversed exactly once.

Assume there are edges which R traverses more than once, and let $u \xrightarrow{e} v$ be the first such edge on the route of R. By the argument above, $u \neq \rho$. Since the pebble is moved in a round robin fashion, the number of times R has left u, at the time it traverses e for the second time, is $d(u) + 1$. R has not started in u and is not at u now. Thus, u must have an incoming edge which R has traversed more than once before traversing e the second time. This contradicts the choice of e. □

The following simple observation is used in the proofs of the next three lemmas: The exits of every vertex v are used by Preparatory in consecutive order. Exits $v.1, v.2, \ldots, v.l_1$ are used in the first exploration which passes through v. Next, $v.(l_1 + 1), v.(l_1 + 2), \ldots, v.l_2$ are used in the second exploration which passes through v, etc.

Lemma 2 *Let u be a vertex and C_i, $i \geq 1$, is the cycle of the first exploration to pass through u. If during Scan, R traverses all edges of C_i which exit from u, then R has traversed all outgoing edges of u.*

Proof: If all outgoing edges of u belong to one exploration then the claim is trivial. Otherwise, Preparatory places the pebble at the first exit of the second exploration which passes through u. Thus, R has traversed all other outgoing edges of u before it traverses the edges of C_i. □

Lemma 3 *During Scan, R traverses all edges of G.*

Proof: Consider the cycles traced during Preparatory, and let C_i be the cycle traced in the i-th exploration. Assume there are edges of G which are not traversed by R during Scan, and let C_k be the first cycle to contain such an edge. Let s be the vertex in which the k-th exploration has started (and ended).

First, consider the claim that all outgoing edges of s are traversed by R.

If $k = 1$ then the claim follows from the halting condition of Scan. If $k > 1$ then there are earlier cycles which pass through s, and all outgoing edges of these cycles have been traversed by R. By Lemma 2, all outgoing edges of s have been traversed. This proves the first claim.

Now, let us call all edges of C_k, which have been traversed by R, *black*, and the remaining edges of C_k — *red*. The edges of other cycles remain colorless.

Next, consider the (second) claim that there is a vertex on C_k for which the number of black outgoing edges is greater than the number of black incoming edges. Note that this holds if and only if there is a vertex on C_k for which the number of black outgoing edges is less than the number of black incoming edges.

Clearly, in every vertex the exits to black edges are consecutive, and the red ones succeed the black.

Now, let us trace the edges of C_k in the order they have been chosen in Preparatory. According to the first claim, all outgoing edges of s have been traversed by R, and therefore are all black, and thus, the first edge of C_k is black. Consider the first red edge $u \xrightarrow{e} v$ while we trace C_k. Since we have entered u via a black edge, and all our previous entries into u have been via black edges, the number of black incoming edges is an upper bound on the number of times we visited u. All previous exits from u were through black edges, and e is red. Thus, the number of black outgoing edges is less than the number of black incoming edges. This proves the second claim.

Assume v is a vertex on C_k which has more black outgoing edges than black incoming edges.

If C_k is the first cycle which passes through v then the number of times that R has visited v is equal to the number of black outgoing edges, plus the number of colorless outgoing edges; these belong to later cycles, and they have all been traversed by R. This follows from the fact that the pebble has been put, by Preparatory, at the first such exit, and all these exits have been used, by R, before the exits of C_k, while at least one black edge exists. Note that the number of entries by R, when it leaves through the last black outgoing edge, is equal to the number of departures. Also note that the number of colorless incoming edges is equal to the number of colorless outgoing edges and that no edge is traversed more than once (Lemma 1). It follows that the number of black incoming edges must be greater than or equal to the number of black outgoing edges. A contradiction.

If C_k is not the first cycle which passes through v, then the choice of C_k and Lemma 2 imply that all outgoing edges have been traversed by R, and therefore all incoming edges of u have been traversed as well. Therefore the number of black incoming edges is equal to the number of black outgoing outgoing edges. A contradiction.

It follows that every edge is traversed by R. □

Theorem 2 *The robot R, moving according to algorithm Scan, traverses an Eulerian cycle.*

Proof: By Lemma 1 no edge is traversed more than once, and by Lemma 3, every edge is traversed. □

Note that after Scan halts, the pebbles are back where Preparatory had put them.

6 The Advanced Algorithm

In this section we describe an algorithm in which the operation of Preparatory is performed by a traveling robot R, whose control is a finite automaton. This new algorithm will ensure that a pebble is placed on edges in a manner consistent

with Preparatory. In total four pebbles are used (equivalently, the amount of memory required in each vertex v is $4 \log d(v) + O(1)$), and each edge is traversed 3 times.

Each vertex has four pebbles, named *start-edge*, *explore*, *retrace* and *scan*. It is assumed that all four pebbles are initially placed at the same exit. The start-edge pebble will never move. Thus, in what follows, only the movements of the other three pebbles will be discussed.

In addition to the root flag, which marks the initial vertex ρ (whose placement will not be discussed any further), there are two more flags stored in each vertex v:

- The flag *visited* indicates the number of cycles (explorations) which have passed through v; its value is 0, 1, or 2. Initially *visited = 0*. If its value is 2 then there are at least two cycles which have passed through v.
- The flag *seen* indicates if v has previously been passed in the current cycle (exploration). Initially, $seen = 0$, and when R reaches v for the first time during the current exploration, it changes the flag *seen* to 1. Once all edges of an exploration are found, R retraces the cycle and assigns all *seen* flags on the cycle to be 0 again. Thus, the *seen* flag of a vertex v is changed to 1, and back to 0, as many times as there are cycles which pass through it.

The main procedure, which governs the actions of R, is **create-eulerian-cycle**; see Table 3. It employs two subroutines, **explore** (Table 4) and **retrace** (Table 5). Also, we use the macro:

- **exit(X-pebble)**: R exits the vertex it is at via the exit in which the X-pebble is located now, but first it moves the X-pebble to the next exit, in a round-robin fashion.

6.1 The Explore Subroutine

Subroutine **explore** is activated in order to perform an exploration.

If $seen = 0$ then the current vertex is visited during this exploration for the first time. If in addition, $visited = 1$ then the running exploration is not the first to pass through the current vertex, and the scan-pebble is moved to the exit where the explore-pebble is at; which is the first exit of the second exploration to pass through the vertex. If $visited < 2$ then it is incremented. Now *seen* is set to 1. R leaves the current vertex through the exit indicated by the explore-pebble, not before moving it to the next exit, in a round robin fashion.

If $seen = 1$ then inquire whether all outgoing edges of the current vertex have been explored; this is done by checking if the explore-pebble is back where the start-pebble is at. If so, R is back at the vertex where the exploration has started and the subroutine is terminated. Otherwise, there are still unexplored outgoing edges and the exploration continues.

```
1    call explore
2    call retrace
3    repeat
4         exit(scan-pebble)
5         if explore-pebble not at start-edge then do
6              call explore
7              call retrace
8    until R is at ρ and scan-pebble is at start-edge
```

Table 3: Procedure **create-eulerian-cycle**

```
while TRUE
    if seen = 0 then do
        if visited = 1 then
            put scan-pebble at explore-pebble's exit
        if visited < 2 then
            visited ← visited + 1
        seen ← 1
        exit(explore-pebble)
    elsif explore-pebble at start-edge pebble then
        EXIT explore subroutine
    else
        exit(explore-pebble)
```

Table 4: Subroutine **explore**

```
while seen = 1 or retrace-pebble not at start-edge do
    seen ← 0
    exit(retrace-pebble)
```

Table 5: Subroutine **retrace**

6.2 The Retrace subroutine

Subroutine **retrace** is activated after an exploration cycle is found. Its purpose is to reset all *seen* flags of the vertices of the newly found cycle to 0. This is done as follows.

If $seen = 0$ and the retrace-pebble is at the start-edge, then all edges of the new cycle have been retraced and **retrace** halts. The halting condition can occur only at the vertex where the exploration has started and all exits of that cycle have been retraced. In every encountered vertex *seen* is set to 0, whatever its value has been.

6.3 The Create-Eulerian-Cycle Procedure

We shall refer to lines 3,4 and 8 of Table 3 as the Scan behavior.

Procedure **create-eulerian-cycle** uses **explore** and **retrace** to perform a deterministic version of Preparatory. R starts an exploration from ρ, first performing **explore** and then **retrace**. Now it is back at ρ, while all three pebbles are at the start-edge. R behaves now as Scan, advancing the scan-pebble only, but this behavior is stopped when R hits a vertex v which has new outgoing edges; this is detected by the fact that the exit where the explore-pebble is at is not at the start-edge. When this happens, R suspends the Scan behavior. It performs an exploration from v, again by first running **explore** and then **retrace**. While doing that, the scan-pebble of every vertex encountered on this cycle is moved from its start-edge to the first exit of the new exploration if the present exploration is the second to pass through that vertex. When **retrace** terminates, R resumes the Scan behavior.

It is easy to see that **create-eulerian-cycle** causes R to place the scan-pebbles in accord with Preparatory, and when R uses the exits where the scan-pebbles are at, it traverses every edge once. When **create-eulerian-cycle** terminates, the scan-pebbles are back where Preparatory could have placed them. Thus, at that time, one can use Scan behavior to traverse an Eulerian cycle.

References

[1] Y. Afek and E. Gafni, *Distributed Algorithms for Unidirectional Networks*, SIAM J. Comput., Vol. 23, No. 6, 1994, pp. 1152-1178.

[2] S. Albers and M.R. Henzinger, *Exploring Unknown Environments*, SIAM J. Comput., Vol. 29, No. 4, 2000, pp. 1164-1188.

[3] M. Blum and W.J. Sakoda, *On the Capability of Finite Automata in 2 and 3 Dimensional Space*. In Proceeding of the Eighteenth Annual Symposium on Foundations of Computer Science, 1977. pp. 147-161.

[4] X. Deng and C.H. Papadimitriou, *Exploring an Unknown Graph*, J. of Graph Th., Vol. 32, No. 3, 1999, pp. 265-297.

[5] S. Even, *Graph Algorithms*, Computer Science press, 1979.

[6] S. Even, A. Litman and P. Winkler, *Computing with Snakes in Directed Networks of Automata*. J. of Algorithms, Vol. 24, 1997, pp. 158-170.

[7] A. Hemmerling, *Labyrinth Problems; Labyrinth-Searching Abilities of Automata*, Teubner-Texte zur Mathematik, Band 114, 1989.

[8] M.O. Rabin, *Maze Threading Automata*. An unpublished lecture presented at MIT and UC Berkeley, 1967.

[9] R. Tayar, *Scanning Directed Eulerian Mazes by a Finite-State Robot*, Master thesis, Computer Science Department, Technion — Israel Inst. of Tech., Haifa, Israel. Sep. 2000, 31 pages.

Appendix: The Simple Algorithm — Lower bounds and a comparison with depth first techniques

Let us show that, in a certain sense, the upper bound on edge traversals, i.e. $O(|V| \cdot |E|)$, is tight for the Simple Algorithm. And yet, there are some cases in which the Simple Algorithm performs significantly better than any depth-first search algorithm.

We present two examples of graphs, one sparse and one dense, for which the Simple Algorithm reaches its upper bound for edge traversals. As our sparse example, we consider a *simple chain*, as shown in Figure 2. The start exit of every vertex is labeled 1.

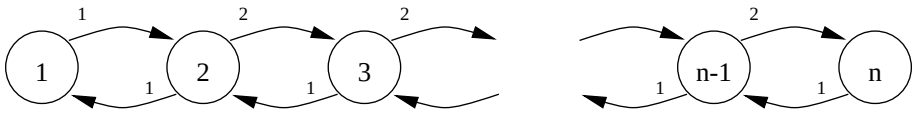

Figure 2: A simple chain

Let us denote by $f_s(n)$ the number of edge traversals until every edge is traversed, when the initial vertex is vertex No. 1. The last edge to be discovered is the edge from vertex n to vertex $n-1$. Clearly, $f_s(2) = 2$ and $f_s(n+1) = f_s(n) + 2n - 1$. Thus,

$$f_s(n) = n^2 - 2n + 2.$$

It follows that in this case, the bound of $O(|V| \cdot |E|)$ is tight.

As our dense graph example, consider the following directed graph $G_p(V, E)$, where p is an odd prime.

$$V = \{0, 1, \ldots, 2p-1\},$$

The set of directed edges consists of two types:

- **body**

  For every two vertices, $i, j < p$, let $\delta(i,j) \stackrel{\triangle}{=} j - i \mod(p)$; i.e. $0 \leq \delta(i,j) < p$. If $\delta(i,j) < \frac{p}{2}$ then there is an edge $i \longrightarrow j$, and the exit at i is labeled $\delta(i,j)$.

- **tail**

  - There is an edge $0 \longrightarrow p$ labeled $\frac{p+1}{2}$.
  - For every $p \leq i < 2p-1$ there is an edge $i \longrightarrow i+1$, labeled 2.
  - For every $p < i \leq 2p-1$ there is an edge $i \longrightarrow i-1$, labeled 1.
  - There is an edge $p \longrightarrow 0$ labeled 1.

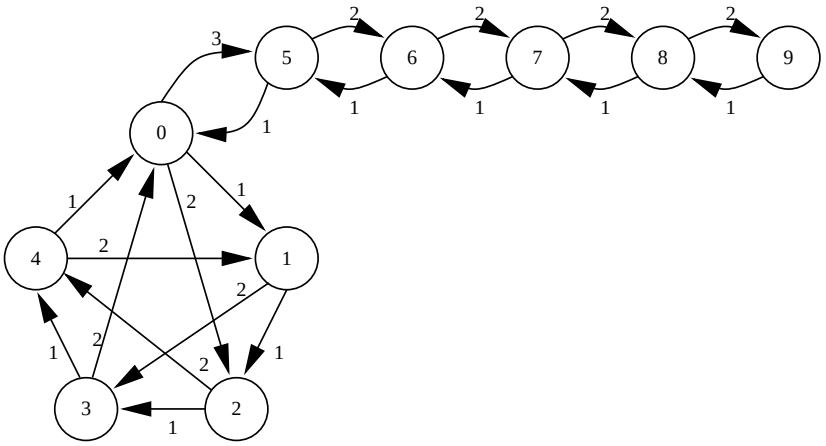

Figure 3: G_5

The case of G_5 is shown in Figure 3.

Note that according to the Simple Algorithm, starting in vertex 0, R scans all $\frac{p(p-1)}{2}$ edges of the body; i.e. the edges between vertices whose name is less than p, before it takes the edge to p. In fact, every time R returns to vertex 0, via the edge from p, it repeats the scan of all edges of the body. This will occur p times. Thus the tour, until the last edge (from $2p-1$ to $2p-2$) is discovered, is of length

$$p \cdot \frac{p(p-1)}{2} + p^2 - 2p + 2.$$

Again, this is of the order $|V| \cdot |E|$.

We now present two examples in which the Simple Algorithm is superior to any DFS algorithm.

The first example is a directed cycle, with a self-loop in every vertex, where exit No. 2 leads to its self-loop. R visits all vertices in the first phase and thus, performs an Eulerian tour on the second phase. The edge traversal complexity is $O(|V|)$. However, if we execute any DFS based algorithm on the same graph, R has to make $\Omega(|V|^2)$ edge traversals.

The second example is the case of a complete directed graph of n vertices; namely for every two vertices a and b there is an edge $a \longrightarrow b$. In the first phase all vertices are visited, since the root has an edge to every other vertex. Therefore, all edges not traversed in the first phase are traversed in the second, and the third phase is the last phase. Thus, the total number of edge traversals is $O(|V|^2)$. However, if one applies a DFS based algorithm, it takes $\Omega(|V|^3)$ steps. (The DFS tree is a directed path of length $|V|-1$, and each time a back-edge is traversed, the tree must be traversed again, at least up to the parent of the start vertex of the back-edge.).