A NOTE ON THE COMPLEXITY OF TRAVERSING A LABYRINTH

Wojciech Rytter

Instytut Informatyki

Uniwersytet Warszawski

00-901 Warszawa

PKiN VII p.

1. INTRODUCTION

When traversing a graph it is frequently necessary to ask "was a given vertex visited before?". If the set of vertices is known in advance and if the graph is labelled (i.e. we know what is i-th vertex) then using an array representation we can answer this question in constant time. However, there are situations when this is not so simple. We present a family of graphs, called plane labyrinths whose vertices are not known explicitely at the beginning of the traversal: Thus, initially we know only the starting vertex and some rules describing the neighbourhoods of vertices.

By a plane labyrinth (briefly labyrinth) we mean a five-tuple L = (*start, east, west, south, north*) where *start* is an integer point of the plane and *east, west, south* and *north* are predicates which, for a given node, v, inform us whether there is a passage from v in the corresponding direction. Imagine a mouse tranversing the labyrinth. It can see only locally and it does not know the global structure of the labyrinth. At any moment, it only knows the part of the labyrinth traversed up to this moment. The labyrinth is an undirected connected graph whose vertices are integer points of the plane and such that if two vertices are adjacent then one can be obtained from the other by a *versor* (unit vector) move. Assume that we know the number of vertices which we denote by n and call the *size* of the labyrinth. Note that the number of the edges of L is $O(n)$. The problem is to visit each node of L. The size of the problem is n.

Suppose that our method of traversing the labyrinth is depth-first search. The following procedure tranverses the labyrinth. The initial value of v (current vertex) is equal to *start*.

```
procedure search;
var d: direction
begin
  for each direction d do
    if open (v,d) then
    begin
      forward move: v:= v+d;
      checking: if not visited before then search;
      return move: v:= v-d
    end
end
```

The procedure requires some comments. The possible values of d are *east, west, south, north*. We can treat them as versors. Hence $-d$ denotes the direction reverse to d. We can treat also every vertex v as a point and the assignment $v:=v+d$ shifts v in the direction d. If we want to implement this algorithm in linear time then we must check that the vertex v was not visited before using only constant time on the average. We denonte by $open(v,d)$ the relation which is true whenever there is a passage from v in direction d.

The total number of executed assignment statements $v:=v+d$ and $v:=v-d$ is linear with respect to n because if the vertex v was visited before then, at this stage, we do not call the procedure *search*. Each of these statements is executed at most once for each value of v and its direction d.

To check quickly if v was visited before we embed the labyrinth L into the bigger labyrinth P. We call P the pyramidal labyrinth. It is not a plane labyrinth but an agglomeration of plane labyrinths.

2. TRAVERSING A PYRAMIDAL LABYRINTH.

For a particular plane labyrinth L of size n. The pyramidal labyrinth (briefly pyramid) P consists of interconnected plane labyrinths $L_0, L_1, \ldots, L_k$, where $k = \log_2 n$ and $L_0 = L$. If $v = (x,y)$ then by $\lfloor v/2 \rfloor$ we denote the point $(\lfloor x/2 \rfloor, \lfloor y/2 \rfloor)$. For $m > 0$ $L_m = (V_m, E_m)$ where V_m is the set of vertices of L_m, $V_m = \{ \lfloor v/2 \rfloor \mid v \in V_{m-1} \}$ and E_m is the set of edges, $E_m = \{ (\lfloor v/2 \rfloor, \lfloor w/2 \rfloor) \mid (v,w) \in E_{m-1} \}$.

Figures 1 and 2 show a plane labyrinth and its pyramid. Observe that L_k has always at most four vertices. In addition to horizontal connections we have also

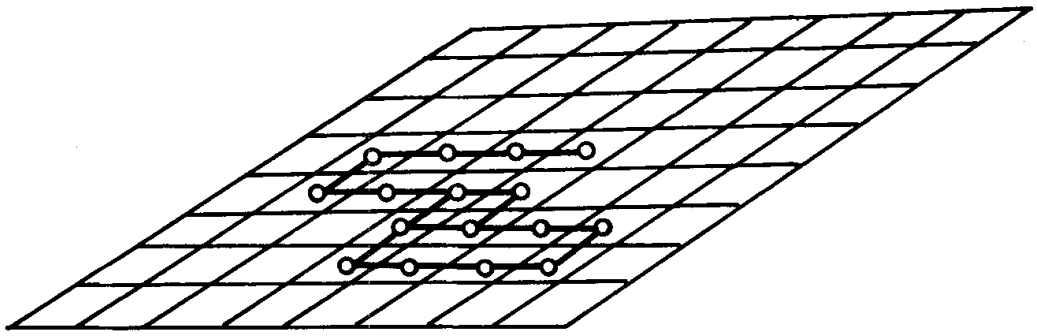

Figure 1. The labyrinth L

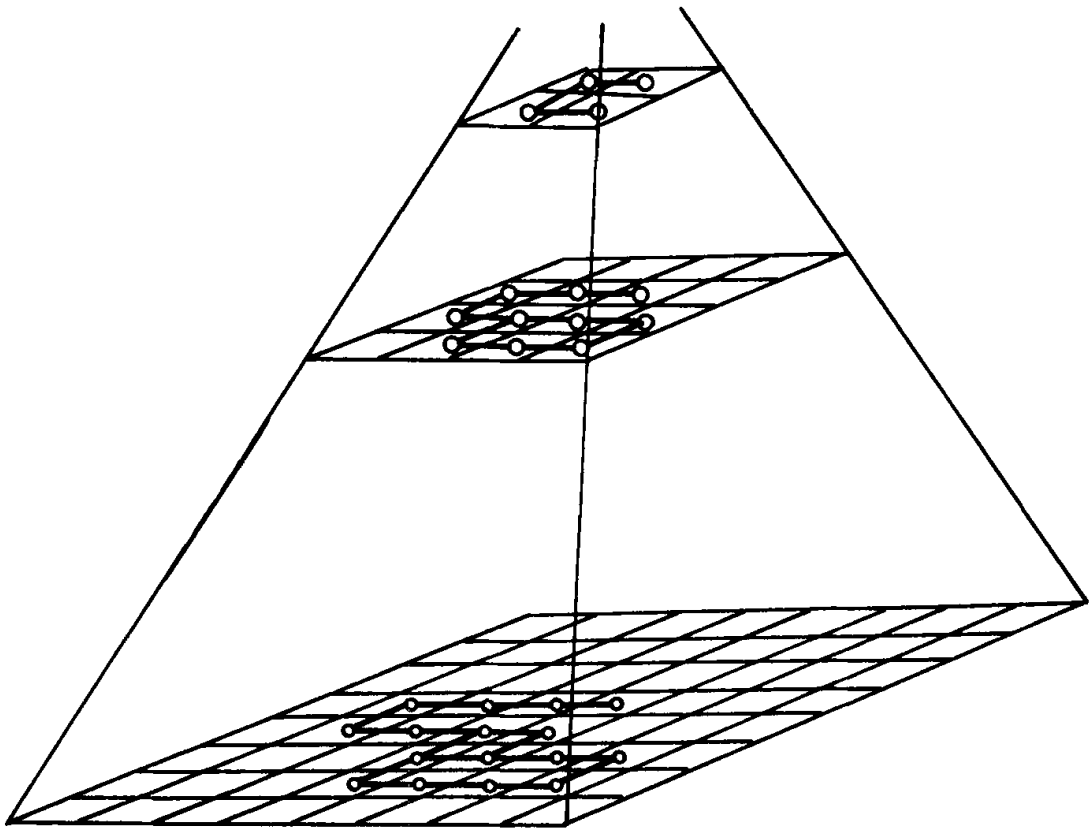

Figure 2. The corresponding to Figure 1 pyramidal labyrinth P

vertical connections (edges) between each v in L_{m-1} and $\lfloor v/2 \rfloor$ in L_m for $m \leq k$. At the outset we know only the starting vertex in the pyramid which is the same as the starting vertex in L. The pyramidal structure was used in automata theory [2] We present here a simpler algorithm which possesses a recursive structure corresponding naturally to the recursive structure of the pyramid.

We design procedure *MOVE* which shifts the vertex v of the labyrinth L_m in the direction d (executing $v:=v+d$) and creates some new nodes in the pyramid. At each stage, upon termination of *MOVE*, the pyramid will contain the part of the labyrinth traversed at this stage, the parts of the compressed labyrinths traversed at this stage and edges of labyrinths and interconnections between plane labyrinths. Assume that at the bottom of the pyramid is L_0 and at the top is L_k. The plane labyrinths are connected in a bottom up manner. The edges leading from $\lfloor v/2 \rfloor$ to v we call down edges. There are at most four down edges from a given vertex. Hence, with every vertex is associated information about the small labyrinth lying below it and containing at most four edges. Hence, trivially there exists a constant time procedure *MOVE1* which makes a move in L_k.

```
procedure MOVE (v,d,m);
var v1: vertex
begin
  if m=k then MOVE1 (v)
  else
    if the edge from v in the direction d is already created
    then  v:=v+d
    else
    begin
      v1:= ⌊v/2⌋ ;
      if  v1 ≠ ⌊(v+d)/2⌋ then  MOVE(v1,d,m+1)
      if there exists a created edge leading from v1 down to v+d
      then find the vertex corresponding to v+d using this edge
      else
      begin
        create a vertex corresponding to v+d;
        visited := false
      end
      v:= v+d;
      create a transversed edge and all necessary connections between
         v and ⌊v/2⌋ if these connections were not created before
    end
end
```

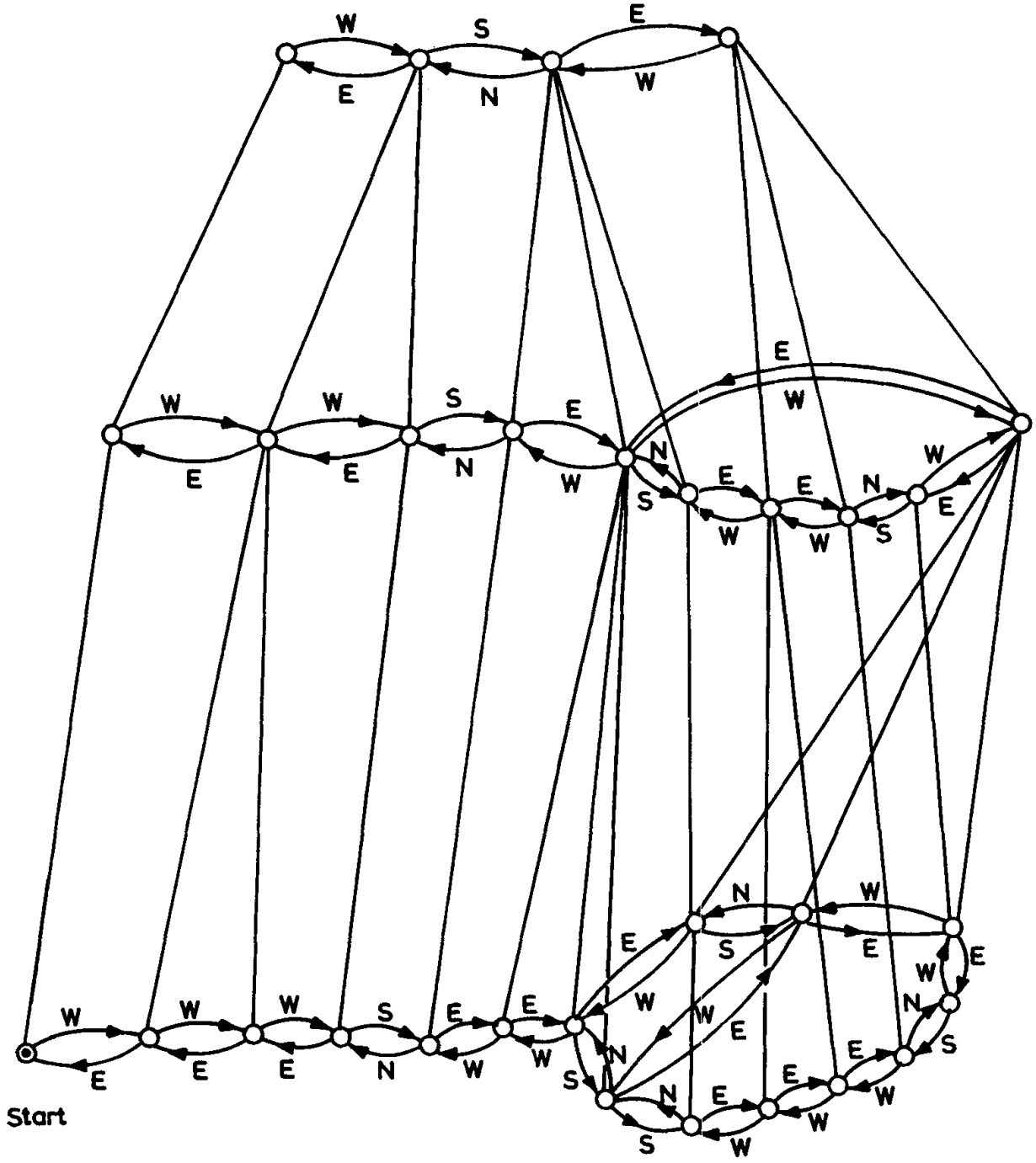

Figure 3. A computer view of the pyramid P. Each horizontal edge is represented by two labelled pointers. The labels *E, W, N, S* denote, respectively, the directions *east, west, north and south.*

In this procedure v has a double meaning. It is a vertex of the labyrinth and it is also the name of a node in the data structure which represents this vertex. The pyramid is created during the traversal. Edges and connections of the pyramid are represented in the data structure created by pointers and this data structure is a computer representation of the pyramid. Figure 3 shows a computer view of the pyramid P. Using *MOVE* we transform the procedure to *search* to the following procedure *SEARCH*.

```
procedure SEARCH;
var d: direction
begin
  for each direction d do
    if open(v,d) then
    begin
      visited:= true ;
      forward move:  MOVE (v,d,0),
      checking     :  if not visited then SEARCH;
      return move  :  MOVE (v,-d,0)
    end
end
```

Let $M(m)$ denote the number of calls $MOVE(v,d,m)$ executed.

LEMMA. *Let* $m < k$.

$$M(m+1) \leq a\, M(m) + \text{before some } a < 1 \text{ and a constant } b.$$

Proof.

Note that every five calls to *MOVE* in L_m contain at most four calls to *MOVE* in L_{m+1}. Hence we can take $a = 4/5$ and $b = 4$. This ends the proof.

THEOREM. The time complexity of traversing a labyrinth of size n is $0(n)$.

Proof.

We use the procedure *SEARCH* with v initially equal to the starting vertex. The time complexity is proportional to the number of calls to the procedure *MOVE*.

$$T(n) \leq c \sum_{i=0}^{\log n} M(i) \leq c\, M(0) \sum_{i=0}^{\infty} a^i + cb \log n = 0(n)$$

because in L_0 we traverse each edge at most twice and we have $0(n)$edges. This completes the proof. ∎

Imagine again a mouse traversing the labyrinth. It can not contain in its memory

the square containing all the labyrinth because this leads to $O(n^2)$-time algorithm. The pyramid from the point of view of the mouse (or a computer) consists only of the vertices and edges created. In this way we have proved that the size of the pyramid P is $O(n)$. Let $P(n)$ denote the number of vertices of the pyramid P. What is the minimal constant c such that $P(n) \leq cn$?

REFERENCES.

[1] A. Aho, J. Hopcroft, J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley (1974).

[2] A. Schonhage, *Storage modification machines*, SIAM on Comp., (1980).