7

Depth-First Search (Ariadne & Co.)

Michael Dom, Falk Hüffner, and Rolf Niedermeier

Friedrich-Schiller-Universität Jena, Jena, Germany
Humboldt-Universität zu Berlin, Berlin, Germany
Technische Universität Berlin, Berlin, Germany

> "Now this happens to those who become hasty in a maze: their very haste gets them more and more entangled."
>
> Lucius Annaeus Seneca (4 BC – 65 AD)

Ariadne, who according to Greek mythology was the daughter of Minos, the king of Crete, fell in love with Theseus. This Athenian hero had been entrusted with killing the Minotaur, a monster half man and half bull. The challenge was made vastly more difficult by the fact that the Minotaur was hidden in the Labyrinth. The clever Ariadne provided her hero with a ball of thread: by fixing the end of the thread at the entry of the Labyrinth and unrolling the thread while traversing the Labyrinth, Theseus could, on the one hand, avoid searching parts of the Labyrinth repeatedly, and, on the other hand, be sure to find his way back into Ariadne's arms.

Not just the ancient Greeks had to deal with the efficient search of spaces such as labyrinths; this task also plays a central role in computer science. One method for this is *depth-first search*, which we examine more closely in the following.

Algorithmic Idea and Implementation

As already mentioned, the problem is to completely search a labyrinth. Here, a labyrinth is a system of corridors, dead ends, and junctions, and the task

B. Vöcking et al. (eds.), *Algorithms Unplugged*,
DOI 10.1007/978-3-642-15328-0_7, © Springer-Verlag Berlin Heidelberg 2011

is thus to visit every junction and every dead end at least once. Further, we would like to pass each corridor no more than once in each direction – after all, Theseus needs to have enough strength in the end for both the Minotaur and Ariadne.

Probably the simplest idea to solve this problem is to just walk into the labyrinth from the starting point and to tick off each junction as it is encountered. If you wind up in a dead end or a junction you have seen before, you turn around, go back to the last junction, and try again from there in another, still unexplored direction. If there is no unexplored direction, then go back to another junction and so on.

Does this method actually lead to the goal? Let us look at the search in more detail; to simplify the description, we use a piece of chalk instead of a thread. With the chalk we mark at each junction the outgoing corridors, with one tick for corridors previously traversed, and with two ticks for corridors traversed twice (that is, in two directions). Specifically, the rules for our search in the labyrinth are as follows.

- If you are in a dead end, turn around and go back to the last junction.
- If you reach a junction, tick the wall of the corridor you came from to be able to find the way back later. After this, there are several possibilities:
  1. First, you check whether you moved in a circle: If the corridor you came from just got its first tick, and there are also ticks visible on other corridors of the junction, then this is the case. You then make a second tick on the corridor you came from and turn around.
  2. Otherwise, you check whether the junction has unexplored corridors: If there are corridors without ticks, then choose an arbitrary one (say the first to the left), mark it with a tick and leave the junction through this corridor. (Incidentally, this is the case at the start of the search.)
  3. Otherwise, there is at most one corridor with only one tick, and all other corridors have two ticks. Thus, you have already explored all corridors leaving the current junction, and leave through the corridor with only one tick, giving it a second tick as a matter of form. If there is no such corridor, that is, all corridors already have two ticks, then you are back at the start and have completely searched the labyrinth.

Let us now look at the example shown in Fig. 7.1, where a path from the start A to the goal F is sought. (That is, again we must traverse the entire labyrinth, but the search can be cut short when F is found.) We assume that a dead end can be recognized as such only upon reaching it.

You start from A northwards. The first junction is C. There you leave a tick at the southbound exit (1). Of course there is no other tick here, so you choose the first unmarked corridor to the left, which is the one toward the west, and tick it (2). Then you reach a dead end at B and turn around. Back at C, the westbound corridor has now two ticks, the southbound one, but the northbound is not marked at all. Thus, you choose this way. At E, there is again an unexplored junction, and from the three possible corridors you

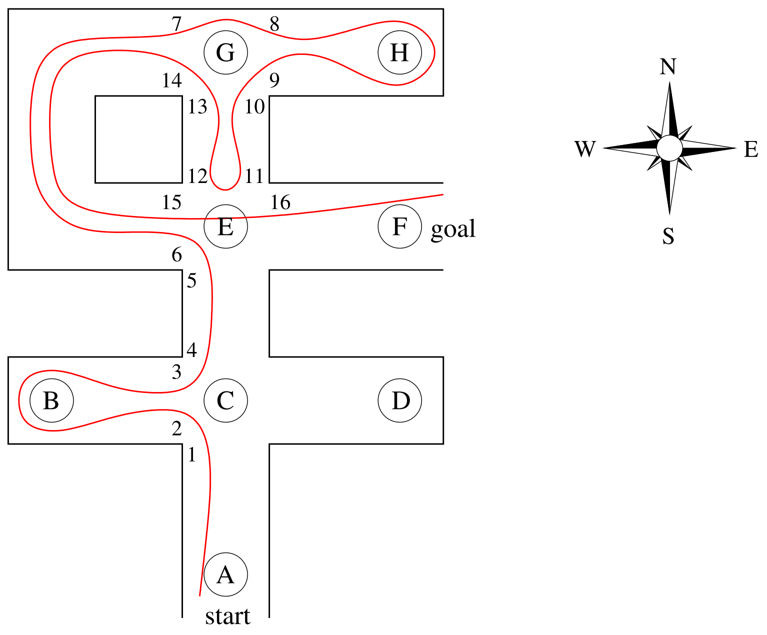

Fig. 7.1. Example for depth-first search in a labyrinth. Starting from A, a path to F is sought. Numbers mark the places where chalk ticks are left

choose the one towards the west. After two turns, you cross straight ahead over the junction at G, leaving two ticks behind (7 and 8). In H, you reach a dead end, so you again turn around. In G, there is only one option left: southwards to E. Here the rule against running in circles comes into effect for the first time: On entering E, you made a tick at the northbound exit of E (11); further, there is a tick at the southbound exit (5) and one at the westbound exit (6) – thus, you need to turn back. Over the junction G and two curves you go back, so that the northern part is now completely searched and you are back at E. Towards the east, there is no tick, and you go there. Finally, you reach the goal at F.

The principle we have learned here is called *depth-first search*, since as described we always go as deep as possible into the labyrinth and only turn around when it is not possible to proceed or a known place is encountered. Only in these cases do we go back a bit and try again from an earlier point into another direction.

The rules for depth-first search are so simple that they can be taught to a computer with only a few lines of code. For each junction, a "state" is stored, and initially the states of all junctions are set to "undiscovered." When the DepthFirstSearch function is called at a junction X, it is first tested whether we moved in a circle (line 2 in the program fragment DepthFirstSearch I shown in Fig. 7.2). Next, it is checked whether the goal was

```
DepthFirstSearch I
1  function DepthFirstSearch($X$):
2     if $state[X]$ = “discovered” then return; endif
3     if $X = goal$ then exit “Goal found!”; endif
4     $state[X]$ := “discovered”;
5     for each neighboring junction $Y$ of $X$
6        DepthFirstSearch($Y$);
7     end for
8  end function     // End of DepthFirstSearch function
9  DepthFirstSearch($start\_junction$);     // Main program
```

Fig. 7.2. Program code for depth-first search using recursion

reached (line 3) – if so, the program quits with the “exit” command, and the search is finished. Otherwise it goes on, and the junction X is marked as “discovered” (line 4). Now, all neighboring junctions that have not been explored yet need to be visited. To do this, the DepthFirstSearch function calls itself for each neighboring junction Y (lines 5–7). This is a frequent trick in programming called *recursion*, which was already described in Chap. 1. When the newly called DepthFirstSearch function notices that Y was already visited and we thus moved in a circle, it immediately returns (line 2) to the calling function at the junction X. Otherwise, the search continues at junction Y.

Sometimes, one wants to avoid recursion, one reason being that at each recursive call in the computer implementation additional time is spent to allocate variables etc. In this case, the depth-first search can be coded without recursion using a *stack*. A stack is a data structure that allows placing objects (in our case junctions) on top of the stack or to remove the object currently on top of the stack. For us, the stack serves to store the return path; we always put a junction X on top of the stack when leaving it, together with a number “exits” that indicates how many of the corridors leaving X have already been explored (see Fig. 7.3). For each junction, we have an array listing all neighbor junctions – thus, we can easily retrieve, e.g., the fifth neighbor junction of a junction X when needed. The variable “mode” contains the information whether we are following an unexplored corridor or whether we are coming back from an already explored junction, going through a corridor that we have already passed in the other direction.

Applications

The depth-first search method works not only for labyrinths, but it also has applications in completely different contexts, as we see in this section.

```
DEPTHFIRSTSEARCH II
$X$ := $start\_junction$;   $mode$ := “forwards”;
repeat
    if $mode$ = “forwards” then
        // we came here through a new corridor
        if $state[X]$ = “discovered” then
            $mode$ := “backwards”;
            take the top pair ($X$, $exits$) off the stack;
        else      // junction is unexplored so far
            if $X = goal$ then exit “Goal found!”; endif
            $state[X]$ := “discovered”;
            if $X$ has no exits then exit “Goal not found!”; endif
            put the pair ($X$, 1) on top of the stack;
            $X$ := the first neighboring junction of $X$;
        endif
    else      // we are coming back
        if $exits$ < number of neighboring junctions of $X$ then
            $exits := exits + 1$;
            put the pair ($X$, $exits$) on top of the stack;
            $mode$ := “forwards”;
            $X$ := neighboring junction number $exits$ of $X$;
        else      // there are no unexplored corridors here any more
            if the stack is empty then
                exit “Goal not found!”;
            else      // we go back further
                take the top pair ($X$, $exits$) off the stack;
            endif
        endif
    endif
end repeat
```

Fig. 7.3. Program code for depth-first search without recursion

Example: Web Search

Here, we do not consider Theseus who is wandering about in the labyrinth, but instead observe a student called Sinon who is searching for a specific Web page.

Sinon Davis has recently been to a party given by his classmate Ariadne, and there he struck up a conversation with a cute girl. Now he would like to see her again, but unfortunately he has not asked for her name. What to do? Of course, Sinon could ask Ariadne, but first he is too shy, and second Ariadne herself does not know all the guests from her party. Finally, Sinon has a bright idea: Why not look up the girl on the Internet site “fazebook.org”? In this commonly known social network platform, almost every young person in the country has a profile, typically with a photo and with links to profiles of

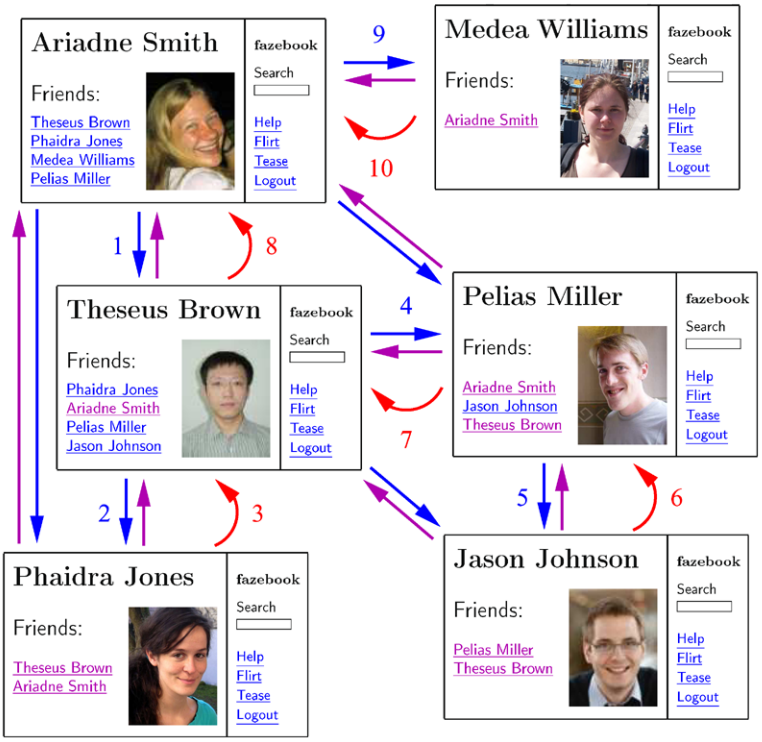

Fig. 7.4. Depth-first search in fazebook: The numbers show the order of the hops from profile to profile. A straight arrow means that a profile contains a link to another profile. A curved arrow means that the "Back" button is used at this place. Note that not all shown links are used since some of them are displayed in violet at the moment when the profile is visited

friends. Hence, Sinon could just visit Ariadne's profile in fazebook and, starting from there, go through the profiles of all her friends, friends of friends, friends of friends of friends, and so on – until he has either found his adored (hopefully, she has a photo in her profile) or completely scanned the relevant parts of Ariadne's surrounding. Sinon, therefore, is confronted with the following task: scan all profiles in fazebook that are reachable from Ariadne's profile via links whose owners have been at the party. As in the previous section, the difficulty is again, on the one hand, not endlessly to move in a circle and, on the other hand, to check everything completely and systematically. This can be done efficiently with a depth-first search in the network of the profiles in fazebook.

Thus, assume that Sinon is beginning his search and is starting at Ariadne's profile. Hence, he clicks on the first link in the list of Ariadne's friends and arrives at the profile of Theseus (Fig. 7.4). Since Theseus is not the person Sinon is searching for, Sinon continues with clicking on the first link of his profile. In this way, he follows the links from profile to profile, always taking care not to follow links that lead to profiles that he has already visited. Sinon, a skilled Web surfer, can recognize these links because his browser displays them in violet instead of blue (this helpful function of the browser in a sense corresponds to the chalk markings in our first example). When Sinon finally reaches a profile whose owner has not been at the party, or when all friends of the owner have already been processed (and displayed in violet), then Sinon clicks on the Back button of his browser and continues with the friends on the resulting profile. Like DepthFirstSearch II, the Back function of the browser uses a stack; whenever a link is clicked, the address of the page containing the link is put on the stack. Furthermore, whenever the Back button is used, the browser jumps to the address on the top of the stack and removes the address from the stack.

If Sinon's adored one has a photo on her profile, and if her profile can be reached from Ariadne's profile by following a series of links whose owners were all at the party (which was our assumption), then Sinon will definitely find her with this method! If, however, Sinon finally ends up on Ariadne's profile by clicking on the Back button, and all links on Ariadne's profile have been visited (and, hence, displayed in violet), then this means that Sinon has bad luck and will not find her – but at least he can be sure that he has not missed any of the profiles coming into question.

Example: Labyrinth Creation

Depth-first search is useful not only for Theseus, but also for the Minotaur: it can be used to create very confusing labyrinths. The method is very simple: Take a regular grid that consists of squares that are separated by "borders", and start the depth-first search at an arbitrary square of the grid. Then, the depth-first search recursively calls itself for all neighboring squares *in random order* (for example, the random number algorithm from Chap. 25 can be used here). Whenever a square is visited for the first time, the border to the preceding square is destroyed (that is, the border to the square from which the depth-first search reached the new square). The result is a pattern like the one shown in Fig. 7.5. Since the depth-first search visits each square, it creates a path from the starting point to each square and, therefore, from each square to each other square. However, these paths are not easy to discover.

Example: Television Shows

Let us assume that the television show "Sick Sister" is going to produce two new seasons. In this show, the candidates are put into the "Sick Sister House," where they are observed by cameras the whole day (and night). In order to provide interesting entertainment, the candidates should be at loggerheads as often as possible, which means that in each season there should be no two candidates in the house who like each other. Now assume that the candidates

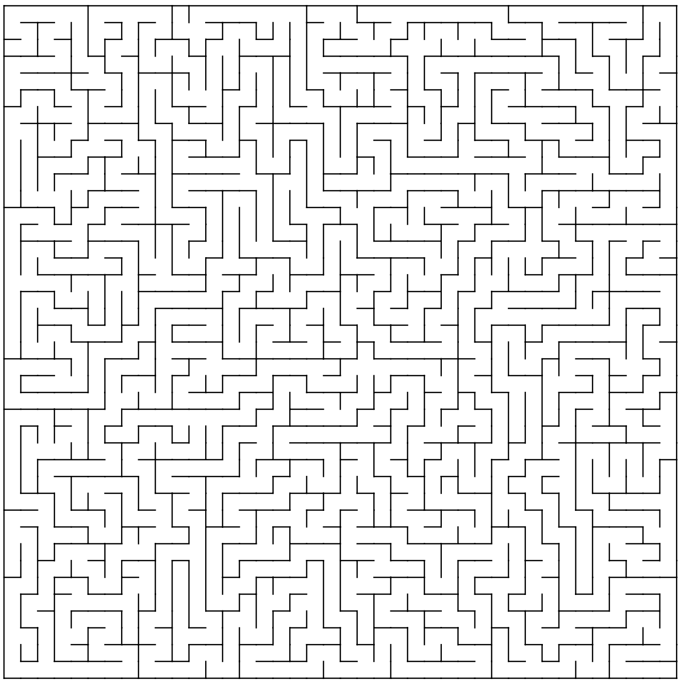

Fig. 7.5. Example for a labyrinth created by a depth-first search

for the upcoming two seasons have already been selected, but for each candidate it still has to be decided – of course, obeying the rule "no two candidates in the house that like each other" – whether he (or she) is part of the first or the second season.

To solve this task, one can create a "sympathy graph": on a piece of paper, draw a small circle – called *vertex* – for each candidate, and connect two circles with a line – called *edge* – if the corresponding candidates like each other (see Fig. 7.6 for an example).[1] Now, the task is to color the vertices of the graph in such a way that not more than two colors are used, each vertex gets exactly one color, and no two vertices that are connected by an edge get the same color. Once the graph is colored, the color of each vertex tells in which season the corresponding candidate has to take part. The desired "two-coloring" for the graph can be found using a depth-first search: Choose an arbitrary vertex and arbitrarily assign one of the two allowed colors to it. Then start the depth-first search at this vertex. Whenever the depth-first

[1] This type of graph, consisting of vertices and edges, has nothing to do with graphs of functions as they are known in the field of mathematics called analysis. "Vertices-and-edges graphs" can be used to model a variety of circumstances and objects, for example, labyrinths: each dead end and each crossing of the labyrinth can be modeled as a vertex and each path of the labyrinth as an edge.

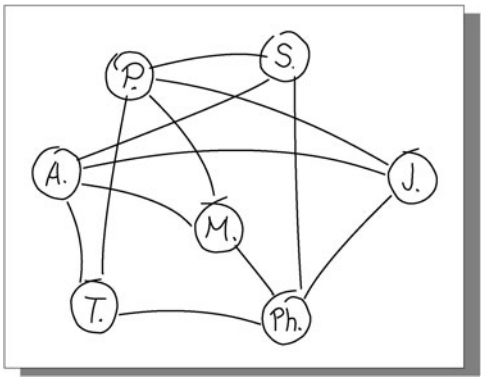

Fig. 7.6. A "sympathy graph": If two persons like each other, then the corresponding vertices (marked with the initials of the persons) are connected by an edge

search comes from a vertex X to a vertex Y that has not been visited before, Y gets the color that X does not have. If the depth-first search comes from a vertex X to a vertex Y that has already been visited, check whether X and Y have different colors. If this is not the case (that is, two vertices of the same color have been found that are connected by an edge), then it is not possible to color the graph as desired. Otherwise, the depth-first search will yield the coloring and provide for exciting entertainment with "Sick Sister."[2]

We mention in passing that graphs that can be colored with two colors as described are called *bipartite*; they are exactly those graphs that do not contain a cycle of odd length. If the candidates shall be distributed among three seasons instead of two, then the task is much more difficult and nobody knows whether it can be solved efficiently with a depth-first search. (The problem is that whenever one reaches a vertex that has not been visited before, one has the choice between two colors and does not know which of them to take.)

Example: Traffic Planning

A further application of depth-first search reads as follows. For the sake of traffic calming in his town, councilman Hermes wants to declare a number of streets to be one-way streets. Thereby, Hermes has to beware of incurring the wrath of the car drivers: He may only prune the street network in such a way

[2] In the special case where the sympathy graph consists of several parts – called "connected components" – that are not connected to each other, then the depth-first search has to be executed separately for each connected component. This can even provide a way to distribute the candidates among the seasons as equally as possible (concerning the number of candidates): just exchange the two colors within some of the connected components.

that no isolated "islands" are generated that cannot be entered or left – that is, it must still be possible to drive from each place to each other place. In graph theory, one would say that the street network must remain one single "strongly connected component". Again, this problem can be solved efficiently by using depth-first search; Chap. 9 takes a closer look at this issue.

Breadth-First Search

One problem of depth-first search is that one can quickly move far away from the start. In many cases, however, it is known that the goal is not too far. For example in fazebook, it can be assumed that the wanted profile is at a distance of maybe at most three from the host. In this case, a *breadth-first search* is more appropriate: It searches the graph layer-wise from the start point, that is, first all direct neighbors (distance 1), then all vertices at distance 2, etc. For this, a data structure called "queue" is used (instead of a stack as is used for depth-first search). A queue allows us to append vertices at the end and to remove the vertex that is currently at the front; in case of breadth-first search, whenever a vertex is removed from the queue, one jumps to this vertex. Therefore, breadth-first search is not applicable for searching a labyrinth: One cannot simply note a junction on a list and "jump" to it on demand. For many other applications (such as the Web search), though, this is not a problem. The program fragment shown in Fig. 7.7 shows breadth-first search in detail.

The queue always holds the vertices that still have to be visited. Thus, initially we add the start vertex to the queue (line 2). As long as the queue has not become empty, the following is repeated: The first vertex is taken out (lines 3 and 4), and all neighbors of this vertex are added to the queue

```
BreadthFirstSearch
 1  begin      // initially, the queue is empty
 2     append the start vertex at the end of the queue;
 3     while queue is not empty
 4        take the first vertex $X$ from the queue;
 5        if $state[X] \neq$ "discovered" then
 6           if $X = goal$ then exit "Goal found!"; endif
 7           $state[X] :=$ "discovered";
 8           for each neighboring vertex $Y$ of $X$
 9              append $Y$ at the end of the queue;
10           end for
11        endif
12     end while
13  end
```

Fig. 7.7. Program code for breadth-first search

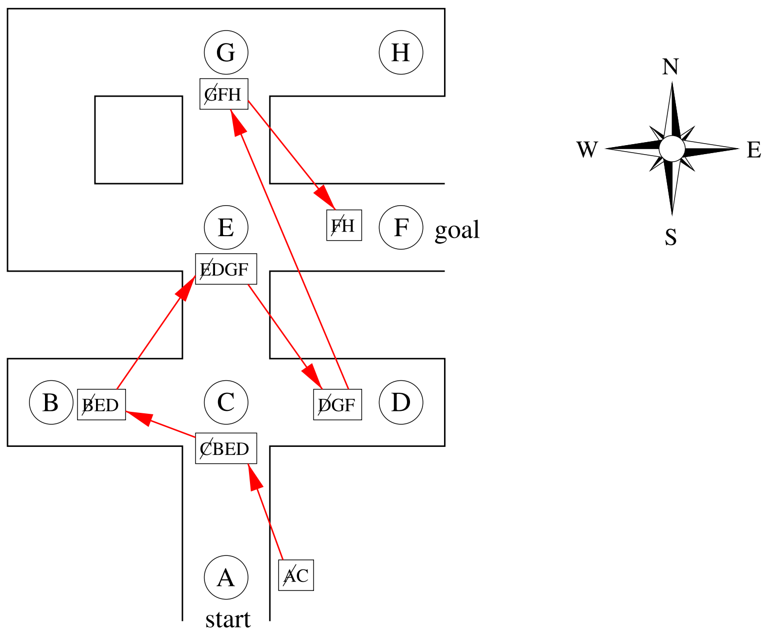

Fig. 7.8. Example for breadth-first search in the labyrinth

(lines 8 and 9). To avoid searching the neighbors of a vertex more than once, whenever a vertex has been treated in this way, it is marked as "discovered" (line 7), and discovered vertices are skipped (line 5).

As an example, we will look at the same graph that has already been used for depth-first search: the labyrinth (Fig. 7.8). Initially, the queue holds only the vertex A. This vertex is taken out, and all neighbors of A are appended; this is only C. The resulting queue is depicted next to A. The search continues with taking out the first element of the queue, which is C. Strictly speaking, all four neighbors of C would now be added to the queue, but as a small optimization we ignore A, since it is already discovered and would not have any effect when taken out of the queue. Thus, B, E, and D are added. At B, no new vertices are added, and we continue directly at E. There, G and F are appended, while C is being ignored. At D, again nothing happens, and we continue at G. Here H is appended, but at F we already find the goal.

It is easy to see that the order in which vertices are visited is very different than with depth-first search (Fig. 7.1). In breadth-first, vertices are visited in order of their distance from the start vertex: first vertex C (distance 1), then B, E, and D (distance 2), finally G and F (distance 3). This also implies that breadth-first always finds a shortest possible path to the goal, and does not consider longer paths in-between.

Incidentally, if a stack is used here instead of a queue, then the algorithm executes a depth-first search instead of a breadth-first search. In contrast to algorithm DepthFirstSearch II, however, the stack is not used to store the return path, but to memorize the junctions to which corridors have been seen but that have not been visited immediately. Therefore, the stack can be significantly larger than with DepthFirstSearch II.

What to choose now with a concrete problem, depth-first search or breadth-first search? Depth-first search is usually slightly easier to implement, since by using recursion, it is not necessary to explicitly maintain a data structure like the queue of breadth-first search. Further, breadth-first search usually uses more memory; for difficult problems, it can even happen that available memory is not sufficient. On the other hand, breadth-first search always finds a shortest path (with respect to the number of edges) and is fast, in particular when the graph is very large but the goal is close to the start. In this case, depth-first search can easily get lost in more distant regions. Thus, which algorithm is the better one depends on the situation at hand.

Further Reading

1. Presentations on depth-first search can be found in most algorithm textbooks.
2. Chapter 9 (Cycles in Graphs)
   In this chapter, another application for depth- and breadth-first search is shown.
3. Chapter 8 (Pledge's Algorithm)
   In our labyrinth example, we assumed that standing at a junction, we can see all exits. But what happens when our torch extinguishes and we are in the dark? Even then it is possible to find the goal; how to do that is explained in Chap. 8.
4. Chapter 32 (Shortest Paths)
   Breadth-first search finds a shortest path if the measure is the number of vertices passed. Often, though, the distances between vertices are different, and we want a path where the sum of the length of the edges is as small as possible. This problem is treated in Chap. 32.

Acknowledgement

We thank Martin Dietzfelbinger (Ilmenau) for many constructive suggestions.

"Everything on earth can be found, if only you do not let yourself be put off searching."

Philemon of Syracuse (ca. 360 BC – 264 BC)