MIT/LCS/TR-536

ALGORITHMS FOR EXPLORING AN UNKNOWN GRAPH

Margrit Betke

March 1992

Algorithms for Exploring an Unknown Graph

by

Margrit Betke

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 1991

Signature of Author____
Department of Electrical Engineering and Computer Science
December 19, 1991

Certified by____
Ronald L. Rivest
Professor of Computer Science
Thesis Supervisor

Accepted by____
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Algorithms for Exploring an Unknown Graph

by

Margrit Betke

Submitted to the Department of Electrical Engineering and Computer Science

on December 19, 1991, in partial fulfillment of the

requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

Abstract

We consider the problem of exploring an unknown strongly connected directed graph. We use the exploration model introduced by Deng and Papadimitriou [DP90]. An explorer follows the edges of an unknown graph until she has seen all the edges and vertices of the graph. The explorer does not know how many vertices and edges the graph has, or how the vertices are connected. At each vertex the explorer can see how many edges are leaving the vertex, but she does not know where they lead to. She chooses one such edge and explores it by traversing it.

Deng and Papadimitriou [DP90] have shown that the graph exploration problem for graphs that are very similar to Eulerian graphs can be solved efficiently. They introduce the notion of deficiency for such graphs to measure the "distance" from being Eulerian and give algorithms that solve the exploration problem for deficiency-one and bounded deficiency graphs.

We review and discuss the problem of exploring an unknown Eulerian graph. Deng and Papadimitriou [DP90] give an algorithm that traverses all the edges in an Eulerian graph. We rederive this algorithm starting from Hierholzer's algorithm that finds an Eulerian tour in an Eulerian graph.

We carefully describe and analyze an algorithm for deficiency-one graphs that combines the two algorithms that Deng and Papadimitriou [DP90] give for this problem. The analysis of the algorithm is based on the analysis of their algorithms. We also briefly discuss the problem of exploring a graph of general deficiency.

Thesis Supervisor: Ronald L. Rivest

Title: Professor of Computer Science

Keywords: Machine Learning, Graphs, Map-Making

Contents

Acknowledgements

Thanks go first to my adviser, Ron Rivest. I am grateful to Ron for giving to me a start on resarch, for being an encouraging teacher, and for arranging my financial support. It is fun to work with Ron. Thanks also go to Charles Leiserson for being an exemplary teacher who cares for his students.

Thanks to Christos Papadimitriou and especially Xiaotie Deng for long and helpful discussions about their research.

Thanks to many friends in the theory group at MIT. Thanks for Be Hubbard for all her help. Special thanks to my current and former office- and roommates. I have always been able to count on their support.

I owe the largest part of my gratitude to my family for their love and support. Thanks to my parents Wilfried and Ludmilla Betke, my sister Elisabeth, and my brother Ulrich.

Finally, I am grateful for the generous financial support provided by the Friedrich-Ebert-Stiftung, the NFS (grant CCR-8914428), the ARO (grant N00014-89-J-1988) and the Siemens Corporation.

Chapter 1

Introduction

1.1 The Problem

Consider the problem of a robot exploring its environment. The robot is equipped with sensors like a camera or sonar that provide information about the robot's environment. Imagine that the robot can identify rooms in a building using its sensors. The robot needs a good model of its environment to perform its various tasks. To obtain this model on its own, the robot walks through the building and determines its floor plan. In each room the robot must make a decision about which door it wants to leave the room by. The robot does not know where the exit leads to until it follows it. The robot explores the building until it learns the floor plan of the building.

We model the problem of a robot exploring its environment as a graph exploration problem: *The explorer follows the edges of an unknown graph until it has seen all the edges and vertices of the graph.*

In this thesis, we study strategies that an explorer - we call her Sacajawea[1] - follows to solve the graph exploration problem. The robot problem introduced above can be modeled with an undirected graph. In this thesis, we are mainly concerned with directed graphs. We assume that Sacajawea cannot simply turn around and go back the way she came from. For example,

[1]Sacajawea was an Indian princess who guided Lewis and Clark in their explorations of the Northwest territories.

directed graphs can be used to model the one-way streets in a city. Sacajawea drives around the city following an exploration algorithm until she has learned the map of the city. The fact that Sacajawea cannot "back up", i.e., she can only follow the streets in one direction, makes the process more difficult.

Another application of the graph exploration problem is the "subway problem": Sacajawea tries to come up with the subway map of a city by riding the trains from one station to another until she has taken every possible train out of every station.

Before Sacajawea starts exploring the environment she does not know how many locations (vertices) and how many paths between the locations (edges) she will encounter. Therefore, the vertex set and edge set of the graph that models the environment are initially *unknown* to her. The learning process begins at a *start vertex.* At each stage of the learning process Sacajawea has a current model of the environment. Sacajawea knows at which vertex she is; she can see the "name" of the vertex. Sacajawea also knows the name of the edges that are going out of the current vertex, but she does not know where they lead to. Sacajawea chooses one such edge and explores it by traversing it. Traversing an unknown edge means that her model of the environment improves. She adds the explored edge to her model. Her current vertex is then the vertex that the explored edge leads to.

When Sacajawea is at a vertex, she can not see how many unexplored edges are going into the vertex. She only knows which of the edges that she has traversed so far are going into this vertex.

We assume that the graph is *finite*, because only a finite number of locations in Sacajawea's environment can be learned in a finite amount of time. We also assume the graph to be *strongly connected.* If it was not strongly connected, the Sacajawea would eventually enter a strongly connected component and could not get out and learn more than that component of the graph. Other information about the structure of the graph may be available a priori to her.

We measure the work that the exploration involves in terms of the number of edges traversed. Any "thinking" on the Sacajawea's part is for free. Traversing the mental model is cost-free; traversing edges in the real graph is what costs. It is easy to design algorithms that run in polynomial time in the number of vertices and edges of the graph. A strategy in which the explorer tries to get from every vertex to every other vertex takes polynomial time in number of vertices in the graph. Therefore, it is crucial to consider the *efficiency* with which the explorer can visit every vertex and traverse every edge.

1.2 The Thesis with a View to the History of the Problem

The problem of a building robot that learns from experience is a major objective in the machine learning research. A number of researchers addressed the problem of inferring the structure of a finite environment from experience using various approaches.

The approach of modeling the environment as a deterministic finite-state automaton has been well studied by the machine learning community. Kearns and Valiant [KV89] show that learning by passively observing the behavior of an unknown automaton is hard. Angluin [Ang86] shows that learning by actively experimenting with it is also hard. However, she gives an algorithm that is combination of active and passive learning which identifies the automaton in time polynomial in the size of the automaton and the length of the longest counterexample. She assumes that the learner has a means of resetting the automaton to some start state. Rivest and Schapire [RS89] show how to remove this assumption, so that the robot can learn the environment in one continuous experiment.

In this thesis we use the graph model of the environment that we described above, and that Deng and Papadimitriou [DP90] introduce. This graph model is easier for the learner than the finite-state machine model because the learner now learns the identity of each vertex she visits, rather than just learning the output value at each such vertex. Since it is easy to design algorithms that run in polynomial time in the number of vertices and edges of the graph, we compare an algorithm that solves the graph exploration problem to the optimal off-line algorithm, which is the algorithm that traverses all edges in a strongly connected directed graph as efficiently as possible (using good luck or prior knowledge of the graph). The ratio of the on-line to the off-line cost is called the *competitive ratio.*

The off-line problem is known as the *Chinese Postman Problem* and was proposed by Mei-ko Kwan in [Kwa62]. Edmonds and Johnson [EJ73] solve the Chinese Postman Problem for an undirected graph by performing an all-pairs shortest path computation, solving a minimum weight matching problem, and finding an Eulerian tour in an (Eulerian) graph. Since the minimum weight perfect matching problem is solvable in polynomial time, it follows that the Chinese Postman Problem is also solvable in polynomial time. Edmonds and Johnson also address the problem in which some of the edges in the graph are directed and some are undirected. They show that the Chinese Postman Problem for directed graphs can be solved in polynomial time using an algorithm that solves the network flow problem.

Deng and Papadimitriou [DP90] give an algorithm that traverses all edges in an Eulerian graph. (They are essentially restating Hierholzer's algorithm [Hie73] that finds an Eulerian tour in an Eulerian graph.) In Chapter 3 of this thesis, we show how Hierholzer's algorithm can be implemented to solve the graph exploration problem for Eulerian graphs.

Deng and Papadimitriou's major contribution [DP90] is that they realize that the graph exploration problem for graphs that are very similar to Eulerian graphs can be solved efficiently. They use a parameterization that they call *deficiency* to express how similar a graph is to an Eulerian graph. The competitive ratio of the graph exploration problem is therefore only dependent on the deficiency of the graph, not on the number of vertices or edges that the graph has. In Chapter 4, we carefully prove properties of graphs of deficiency-d that Deng and Papadimitriou assume in their algorithms.

Deng and Papadimitriou show a lower order bound of $O(d^2/\lg d)$ for the competitive ratio of the graph exploration problem for graphs of deficiency d. A proof due to Elias Koutsoupias [DP] increases the lower bound to $O(d^2/4)$. Deng and Papadimitriou give two algorithms that solve the graph exploration problem for graphs with deficiency one. We combine both algorithms and show in great detail how this deficiency-one algorithm can be implemented, and why it leads to a competitive ratio of four. The analysis of our algorithm is also based on Deng and Papadimitriou's ideas.

In the final chapter, we discuss the graph exploration problem for graphs with general deficiency d.

It is apparent that this thesis is based heavily on the seminal work of Deng and Papadimitriou. The original objective our this research was to simplify and extend their work. However, we found their algorithms and proofs to be exceedingly terse, so we decided instead to provide this careful and detailed re-derivation and analysis of their algorithms for the deficiency-one case. (We have also combined their two algorithms into one, and provide numerous missing details and arguments.) While we had thoughts about doing something similar for their deficiency-d algorithm, we found this to be too complicated for the time we had available (and indeed, some of the arguments and details required for a complete understanding still elude us). It remains as an interesting open problem, we feel, to find a *simple* algorithm and analysis for the general deficiency-d case.

Chapter 2

The Exploration Model

We call the explorer's model of an unknown graph during the exploration the *"partial graph"*. The notion of a partial graph was introduced by Deng and Papadimitriou [DP90]. We discuss implementation issues and define some basic operations that we use heavily in the algorithms later. We also describe how we measure the efficiency of the graph exploration algorithms.

2.1 The Partial Graph

The environment to be learned is modeled by a directed graph $G = (V, E)$, where the *vertex set* V is a finite set and the *edge set* E is a binary relation on V. The model also includes a start vertex s.

For each stage of the learning process the explorer's mental model for the parts of the graph that she has visited so far is described by a *partial graph*:

$$G_p = (V_p, E_p)$$

where $V_p \subseteq V$ and $E_p \subseteq E \cap V_p^2$.

The out-degree *od(v)* of a vertex v is the number of edges directed away from v. When a vertex v is first visited, the out-degree of v is apparent to the explorer. The *partial out-degree pod(v)* of a vertex v is the number of outgoing edges of v in the partial graph. The partial out-degree of a vertex is at most as large as the out-degree: $pod(v) \leq od(v)$.

The in-degree *id(v)* of a vertex v is the number of edges directed into v. The *partial in-degree pid(v)* of a vertex v is the number of edges in the partial graph that are directed into v. The partial in-degree of a vertex is be at most as large as the in-degree of the node: $pid(v) \leq id(v)$.

In general, we use the word "partial" to refer to the partial graph.

At each point in time during the learning process, the explorer is at a *current node* $c \in V_p$. Initially, the current vertex is the start vertex s, and the partial graph G_p is

$$G_p = (\{s\}, \emptyset).$$

At each stage, the explorer can either take an unexplored edge out of c, or she can follow an explored edge out of c. The first step is applicable if $pod(c) < od(c)$, the second if there is an edge $(c, v) \in E_p$.

The learning process terminates when the whole graph is explored, i.e., when the partial graph equals the actual graph.

The graph G is assumed to be *finite* and *strongly connected.* The partial graph, however, need not be strongly connected throughout the exploration. The explorer may not be able to get to every vertex in the partial graph using only edges in the partial graph. This complicates any exploration strategy that we describe in the following chapters.

An edge is either *explored* or *unexplored.* Initially all edges are unexplored. An edge is explored when the explorer traverses it for the first time. The explorer knows of the existence of an unexplored edge (v, w) if she has reached the vertex v. The explorer does not know anything about vertex w until she has reached vertex w. When the explorer explores an edge, she adds it to the edge set E_p of the partial graph.

We say that the explorer *discovers* a vertex when she reaches it for the first time. Whenever the explorer discovers a vertex, she adds it to the vertex set V_p of the partial graph. Having discovered vertex v, the explorer can "see" how many edges are leaving vertex v, so she can determine the out-degree of v. The partial out-degree of v is initially zero. Whenever the explorer explores an edge out of v, the partial out-degree is incremented by one. Eventually all edges out of v are explored, and we say that v is "finished."

A vertex is either *finished* or *unfinished*; a vertex is finished if and only if all of its outgoing edges are explored. Initially, therefore, all vertices are unfinished. This claim depends on the assumption that the graph is strongly connected, and so each vertex has nonzero out-degree (except the trivial case that $G = (V, \emptyset)$).

We want to keep track of the order in which the explorer traverses the edges of the graph. We use "paths" to remember which edges the explorer has taken through the graph.

We define a *path* $x \rightsquigarrow y$ from a vertex x to a vertex y in G to be a sequence $< v_0, v_1, \ldots, v_k >$ of vertices such that $x = v_0$, $y = v_k$, and $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \ldots, k$. The path may also be denoted $v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_k$. We call vertex v_0 the *head* of the path, node v_k the *end* of the path, and edge (v_0, v_1) the first edge on the path. The *empty path* starting and finishing at vertex v is denoted $< v >$.

We say that the explorer traverses path $v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_k$, if she follows the edges $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \ldots, k$ to get from v_0 to v_k.

If the explorer traverses a sequence of unexplored edges and stops in a finished vertex, we call the path that is formed by the visited vertices a *walk*.

2.2 Basic Operations

We use the following notation to denote basic operations on paths like *concatenation* and taking the *prefix* or *suffix* of a path:

- AB denotes that path B is appended to path A.
- $A[..i]$ stands for the prefix $< A[0], \ldots, A[i] >$ of path A.
- $A[i..]$ denotes suffix $< A[i], \ldots, A[l_A] >$, where l_A is the length of path A.
- $A[i..j]$ means the portion $< A[i], \ldots, A[j] >$ of path A.

We describe the implementation of the exploration algorithms in pseudocode. We use pseudocode that is very much like Pascal or C. We provide comments following the symbol "▷".

We call the following strategy for the exploration *greedy*: Whenever the explorer has a choice whether to follow an edge that she has never traversed before or to follow an already traversed edge, she takes the never traversed edge.

Deng and Papadimitriou [DP90] point out that the exploration algorithms that they give are *greedy*. However, the explorer does not follow this greedy strategy throughout Deng and Papadimitrou's algorithms. We choose to distinguish carefully the parts of the algorithms that use the greedy approach described above from the ones that do not. We call the greedy exploration an exploration during a *walk*. Deng and Papadimitrou introduced the notion of a *walk*. We define *walks* as follows.

To *take a walk* from a vertex v means that the explorer starts at v and greedily traverses unexplored edges until she arrives at a finished vertex. If v is initially finished, then the walk has zero length and she arrives at v. If she takes a walk from v and arrives back at v, then she is said to *loop*. If she does not loop, then she *gets stuck* at some vertex $w \neq v$. If she takes a walk from a finished vertex v, the walk is defined to be the empty path $<v>$.

The procedure WALK takes the graph G, the partial graph G_p, and a start vertex v as input. Following procedure WALK, the explorer takes a walk from vertex v until she gets stuck in some finished vertex. The procedure WALK returns path P that traversed during the walk.

```
WALK($G, G_p, v$)
1  $c \leftarrow v$                          ▷ $c$ is the current node.
2  create an empty path $P$ where $P[0] = c$ ▷ $c$ is the first vertex of path $P = <c>$
3  while $c$ has an unexplored outgoing edge $(c, x)$
                                    ▷ while $pod(c) < od(c)$
4        do explore $(c, x)$          ▷ include $(c, x)$ in $G_p$
5           append $x$ to path $P$
6           $c \leftarrow x$              ▷ The new current vertex is $x$.
7  return path $P$
```

The condition in line 3 can be checked by comparing the out-degree of c with its partial out-degree. If the partial out-degree of c is smaller than its out-degree, then c is unfinished and there is an edge (c, x) to some unknown vertex x. In line 4 *exploring* the edge (c, x) means that the explorer traverses edge (c, x) and arrives at vertex x. The partial graph is updated with edge (c, x).

Consider a strategy with the property that whenever the explorer starts traversing unexplored edges she continues doing so until she gets stuck. An algorithm that always follows this strategy is called *walk-based.* The difference between a walk-based and a greedy algorithm is that in a greedy strategy, the explorer chooses an unexplored edge whenever she visits an unfinished vertex. A walk-based algorithm can have instructions like *move along path P.* Following this instruction, the explorer would not leave path P to follow an untraversed edge, if P has an unfinished vertex as required. A greedy algorithm, by contrast, would leave P at the first unfinished vertex.

To *try to finish* v or *work on* v given that the explorer is at v means to take a walk from v. If the explorer loops, then v is now finished and the explorer has succeeded in finishing v. If the explorer gets stuck somewhere else, then v may or may not be finished.

If P is a path $v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_n$, then to *work on* P or *try to finish* P (given that the explorer is at v_0) means to try to finish v_0, then (assuming that she loops) to traverse the edge (v_0, v_1) to v_1, then try to finish v_1, and so, until she tries to finish v_n. If the explorer tries to finish v_i, and she takes a walk of the form

$$v_i \rightarrow w_1 \rightarrow w_2 \rightarrow \ldots \rightarrow w_m \rightarrow v_i$$

that finishes v_i (by looping), then that walk is understood to be *inserted* into the path P before the explorer tries to finish it; it is as if the original path were:

$$v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_i \rightarrow w_1 \rightarrow w_2 \rightarrow \ldots \rightarrow w_m \rightarrow v_i \rightarrow v_{i+1} \rightarrow \ldots \rightarrow v_n;$$

the next vertex to try to finish after v_i is finished is w_1, and so on. We call the operation of inserting the walk $w_1 \rightarrow w_2 \rightarrow \ldots \rightarrow w_m$ into path P *splicing* the walk into the path.

If the explorer never gets stuck while working on a path P, then she has succeeded in finishing each vertex in the path, and so we say the path is *finished.* A path containing unfinished vertices is itself said to be *unfinished.* If the explorer tries to finish P and gets stuck at a vertex x when taking the walk from v_i, $v_i \neq x$, then P is only partially finished. The initial segment from v_0 to v_{i-1} is finished, and the final segment from v_i to v_n is (probably) unfinished. We say that the explorer *created* path $v_i \rightsquigarrow x$ while taking a walk from v_i.

If the explorer then wants to finish the final segment of P, she must get back from x to v_i first. We say that she needs to *relocate.* In the following algorithms, we must specify how to do each such relocation and ensure that it is feasible.

The procedure WORK-ON implements the operation *work on a path.* It takes the graph G, the partial graph G_p, and a path P as an input. The procedure returns a boolean variable *new-path-flag* that is true if the explorer took a walk from a some vertex $P[i]$ on path P and got stuck in a vertex v such that $v \neq P[i]$. The procedure also returns the index i, so that relocation to finish the final segment of P is possible, and it returns the path that is created during the walk. If *new-path-flag* is false, then i is the index of the last vertex on P, and the path that is returned is the last walk that is taken from $P[i]$.

```
WORK-ON($G, G_p, P$)
 1  path-finished ← new-path-flag ← False
 2  $i \leftarrow 0$                                      ▷ $P[i]$ is current vertex on path $P$
 3  repeat $W \leftarrow$ WALK($G, G_p, P[i]$)
 4          if explorer at vertex $P[i]$           ▷ $W$ is a loop back to $P[i]$
 5             then splice $W$ into $P$ at index $i$
 6                  if every vertex on $P$ is finished
 7                     then path-finished ← True
 8                     else traverse edge $(P[i], P[i+1])$
 9                          $i \leftarrow i + 1$
10             else new-path-flag ← True       ▷ explorer is at a partial source or sink
11     until path-finished or new-path-flag
12  return new-path-flag, walk $W$, and index $i$.
```

The explorer starts working on path P beginning at vertex $P[0]$ which is the head of path P in line 2. The explorer takes a walk from a vertex $P[i]$ on path P until she gets stuck. If the created walk W is a loop which means that the explorer is back at vertex $P[i]$ (line 4), the walk W is spliced into path P (line 5) at index i. If the explorer finishes a node, she traverses edge $(P[i], P[i+1])$ (line 8) and works on the next vertex, the new $P[i]$ of the path (line 3).

If every vertex on path P is finished (line 6), the boolean variable *path-finished* is set True, and the procedure WORK-ON(G, G_p, P) terminates having finished path P.

If work on some vertex c does not end at c, but in some vertex v, $v \neq c$, the procedure WORK-ON terminates having created a new path $W = < P[i], \ldots, v >$. The portion $< P[0], \ldots, P[i-1] >$ of path P is finished. The portion $< P[i], \ldots, P[l_P] >$, where $P[l_P]$ is the end of path P, is not finished in general.

2.3 Efficiency Measurements

Our goal is to explore the whole graph efficiently. We measure the work in terms of the number of edges traversed. The *trace* of an edge is the number of times it has been traversed (i.e., the number of times it was traced over). The smaller the sum of the traces of all edges in the graph is, the more *efficient* we say the exploration is.

The optimal *off-line* cost is the number of edges that the explorer traverses to cover every edge of the graph if the explorer had a priori a map of the graph and could plan the most efficient route.

The off-line problem is the same as the *"Chinese postman problem"* proposed by Mei-ko Kwan [Kwa62]. There are different approaches to solve the Chinese postman problem that take into account if the graph is directed or undirected [EJ73].

The *depth-first-search* algorithm, e.g. [CLR90], can be applied to the undirected case. In an undirected graph the explorer can go back where she came from. The depth-first-search algorithm relies on this property that the explorer can back up. The depth-first-search algorithm is an off-line algorithm that can be understood as an on-line exploration. The *on-line* algorithm corresponds to what an explorer can really do. It requires that its decisions can only be based on what it has seen so far (and maybe coin flips). Since the *depth-first-search* algorithm has this property, it can be used to explore an undirected graph. The running time of the *depth-first-search* algorithm is $O(V + E)$. Thus, an undirected graph can be explored in $O(E)$ time.

We give a trivial lower bound for the problem of exploring an unknown graph: Any strategy to explore a graph takes $\Omega(E)$ time, since every edge in the graph has to be traversed once when it is explored.

The ratio of the on-line to the off-line cost is called the *competitive ratio*. It is used to analyze the different strategies and measure how well they do in comparison to the optimal off-line traversal.

Chapter 3

Exploring an Eulerian Graph

As we mentioned in the introduction, the explorer may a priori have some information about the structure of the graph. In the following chapter, we study the case that the explorer has some additional information about the degree of the vertices in the graph: she knows that the strongly connected, directed graph is Eulerian.

Deng and Papadimitriou [DP90] make the observation that the properties of an Eulerian graph lead to an efficient algorithm for the graph exploration problem. This observation is important, because it can be generalized to graphs that are very similar to Eulerian graphs. Deng and Papadimitriou [DP90] invent the notion of *deficiency* based on this observation.

In the this chapter, we show how an algorithm due to Hierholzer [Hie73] that finds the Euler tour of an Eulerian graph can be applied to solve the graph exploration problem for Eulerian graphs.

3.1 Eulerian Graphs

An *Euler tour* of a strongly connected, directed graph G is a cycle that contains every edge of G exactly once. We call a graph that contains an Euler tour an *Eulerian graph.* If the path that the explorer traverses during a walk is a loop, then the path is a cycle. If this cycle contains every edge in the graph, the walk is an Euler tour.

Lemma 1 *If the out-degree of every vertex in a graph is equal to its in-degree, then every initial walk taken from a start vertex s in the graph is a loop.*

Proof: During the walk, vertex s has one more outgoing than incoming traversed edge. Every other vertex v, $v \neq s$, has the same number of traversed incoming and outgoing edges after the explorer has visited the vertex. The explorer cannot get stuck in v, because for every unexplored incoming edge, there is an unexplored outgoing edge, since the indegree of v equals its outdegree. Since vertex s has one more incoming than outgoing untraversed edge, the explorer can only get stuck in vertex s. □

Theorem 2 (Hierholzer) *A directed graph is Eulerian iff the graph is connected and the out-degree of every vertex is equal to its in-degree.*

Proof:
($\Longrightarrow$) When a vertex v is visited during an Euler tour, one incoming and one outgoing edge of v are traversed. Since no edge is traversed more than once, visiting a vertex x times during the Euler tour means that the vertex has x incoming and x outgoing edges. Its in- and out-degree is x. When an Euler tour is started at a vertex s, s has one more traversed outgoing edge than traversed incoming edge during the Euler tour. Since the Euler tour is a cycle, the last traversed edge of the Euler tour is an incoming edge of s. Therefore, the start vertex has equal in- and out-degree, too.

($\Longleftarrow$) A walk started at a vertex s cannot end at any other vertex than s as shown in Lemma 1.

In general, however, the path traversed during this walk is not an Euler tour, because we may not have traversed every edge in the graph. Since the graph is connected, one of the untraversed edges must come out of a vertex that is visited during the walk. Assume that the first such vertex on the cycle created by the initial walk is v and the untraversed outgoing edge is (v, w). Vertex v that has both traversed outgoing and incoming edges, and untraversed outgoing and incoming edges.

We change the walk at v to make the path an Euler tour: we traverse (v, w) and then follow only untraversed edges if there are any. Before a vertex z is visited during the walk from v it has the same number of untraversed incoming and untraversed outgoing edges. After vertex z is visited, the number of untraversed edges that lead into and out of z is smaller, but the number of untraversed incoming edges is the same as the number of untraversed outgoing edges. Vertex

v is the only vertex with one more outgoing traversed edge than incoming traversed edge when the walk starts. Therefore, the walk continues until the last unexplored incoming edge of v is traversed. Since v then does not have any untraversed outgoing edges, we get stuck at v.

We splice this walk into the cycle that was created during the initial walk and follow the path until we reach another vertex v' that has an outgoing untraversed edge. We start another walk along untraversed edges until we get back to vertex v'. Following this procedure, we encounter every untraversed edge of the graph and splice the path on which this edge is into the initial cycle. When every untraversed edge that emanates from a vertex visited on one of the walks is considered, no untraversed edges are left in the graph, because the graph is connected. So the final cycle (after all the other walks are spliced in) is an Euler tour. □

3.2 The Eulerian Algorithm

We restated Hierholzer's theorem, because the proof is constructive, and provides a strategy for an efficient algorithm for finding the Euler tour of an Eulerian graph. The algorithm can be applied directly to the exploration problem as follows.

The explorer takes a walk from the start vertex until she gets stuck. Then she traverses the path created by the walk again and starts to take walks from every unfinished vertex; these walks are spliced into the initial walk. Since the graph is Eulerian, she is guaranteed to loop whenever she takes a walk (Lemma 1). Therefore eventually every vertex in the path is finished and the whole graph is explored.

Given a graph G and a start vertex s, EULERIAN-EXPLORATION traverses every edge in the graph at least once and returns the explored graph.

```
EULERIAN-EXPLORATION($G, s$)
1  path $P \leftarrow$ WALK($G, G_p, s$)
2  $i \leftarrow 0$                              ▷ explorer is at $P[0]$
3  while end of path $P$ not reached yet
4       do $P' \leftarrow$ WALK($G, G_p, P[i]$)        ▷ explorer takes a walk from vertex $P[i]$
5          if $P'$ not an empty path
6             then splice $P'$ into $P$ at $P[i]$
7          explorer traverses edge $(P[i], P[i+1])$
8          $i \leftarrow i + 1$
9  return $G_p$
```

Given a start vertex s, the explorer takes a walk on s until it gets stuck in s. We call the path that is created by this initial walk P. The head of P is s, so the explorer takes a walk on s in line 4. The walk creates an empty path P', since the explorer got stuck in s before. Whenever taking a walk on a vertex creates an empty path, the vertex is already finished.

The first time line 7 is executed, the explorer traverses the first edge $e = (s, v)$ on P, and the index i is incremented. Then the explorer takes a walk from vertex $v = P[1]$. This walk may not be empty. If a path P' is created by the walk, it is spliced into P at $P[1]$ (line 6). The exploration is finished when the end of path P is reached. Note that path P is then an Eulerian tour.

Note that algorithm to explore an Eulerian graph can also be implemented using the procedure WORK-ON that is defined in section 2.2.

```
EULERIAN-EXPLORATION'$(G, s)$
1 path $P \leftarrow$ WALK$(G, G_p, s)$
2 $(new, Newpath, i) \leftarrow$ WORK-ON$(G, G_p, P)$
3 return $G_p$
```

Procedure WORK-ON implements the while-loop of EULERIAN-EXPLORATION. Both implementations of the Eulerian exploration problem assume that every walk taken during the exploration loops. We show below why we can make this assumption when exploring Eulerian graphs. Thus, WORK-ON never returns *new* = **True** in line 2 of EULERIAN-EXPLORATION'. We only call it because of its side-effects on the partial graph. If we insert the text of procedure WORK-ON without the lines that are needed to check if a walk is not a loop, procedure EULERIAN-EXPLORATION' would look very much like EULERIAN-EXPLORATION - only that the while-loop is implemented as a repeat-loop. Therefore, correctness of procedure EULERIAN-EXPLORATION' follows directly from the correctness of procedure EULERIAN-EXPLORATION.

Theorem 3 EULERIAN-EXPLORATION *correctly explores an Eulerian graph and traverses each edge of the graph at most twice.*

Proof: The correctness of the Eulerian algorithm follows from the arguments in the proof of Theorem 2.

Any walk taken in an Eulerian graph creates a cycle, because the out-degree of every vertex is equal to its in-degree. If the first cycle created in line 1 of EULERIAN-EXPLORATION is an Euler tour, the algorithm forces the explorer to traverse the cycle once more, taking empty walks from every vertex on the cycle. When the cycle is traversed completely, the algorithm terminates, and all edges of the graph are traversed.

If the path P created in line 1 of EULERIAN-EXPLORATION is not an Euler tour, some edges of the graph have not been explored. At least one of these edges is an outgoing edge of an unfinished vertex on P, because the graph is connected. The algorithm forces the explorer to take a walk from every vertex on path P. Every walk from a vertex $P[i]$ on path P ends in $P[i]$, because $P[i]$ is the only vertex with one more outgoing traversed edge than incoming traversed edge during the walk (see proof of Theorem 2 ($\Leftarrow$)). If a walk from $P[i]$ is empty, the explorer traverses the next edge of the path, i is incremented, and the next walk is taken from the next vertex $P[i]$ on P.

A walk from $P[i]$ is empty if $P[i]$ is a finished vertex. No more work is needed on a finished vertex, therefore, index i is incremented, and a walk is taken from the next vertex on the path. Thus, the explorer will eventually work on all unfinished vertices on the initial path P.

Any path P' that is created by a walk is spliced into P, so that every unfinished vertex on P' will also be worked on.

The end of path P is reached after working on every unfinished vertex on path P. Thus, when the algorithm terminates, all the vertices in the graph that are connected to path P by unexplored edges at some point during the exploration are spliced into path P and therefore finished. Since the graph is connected, every vertex is considered and therefore, the whole graph is explored (see also proof of Theorem 2 ($\Leftarrow$)).

Every edge in the graph is traversed once when it is explored, and once when it is traversed as an edge on P in line 7 of EULERIAN-EXPLORATION'.

An edge cannot be on path P twice, because it is only inserted into P when it is explored, i.e., traversed for the first time. Thus, every edge in the graph is traversed at most twice. □

The off-line cost for traversing an Eulerian graph is E; the on-line cost for exploring an Eulerian graph is at most $2E$. Thus, the competitive ratio for exploring an Eulerian graph is bounded above by 2. Deng and Papadimitriou [DP90] show that this bound is tight by giving the graph illustrated in Figure 3-1. The cycle C_0 in the graph contains many edges. The cycles $C_1, \ldots, C_4$ only contain three edges. If the explorer does not find the Euler tour during an initial walk, she has to follow the "expensive" cycle C_0 in order to get to the cycles $C_1, \ldots, C_4$.

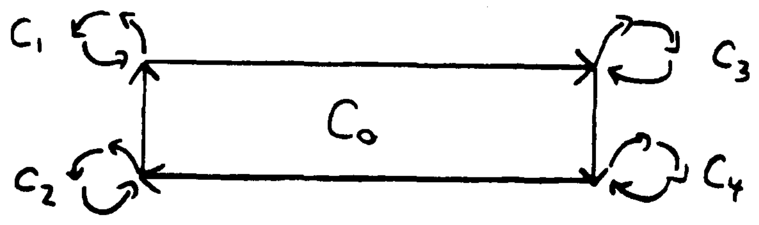


Figure 3-1: Graph that Deng and Papadimitriou use to show the lower bound for the Eulerian exploration problem.

Chapter 4

Deficiency-d Graphs

In the previous chapter we described how the property of being Eulerian gives a very efficient algorithm for exploring a graph. An Eulerian graph is a very special kind of graph. In the following, we generalize the a priori information that the explorer has about the structure of the graph: we allow different out- and in-degrees of the vertices of the graph, but we keep a bound on the sum of these differences. We call this sum the *deficiency* of the graph, a notion introduced by Deng and Papadimitriou [DP90]. The more deficient a graph is, the farther it is from being Eulerian.

4.1 Definitions

The graph has *deficiency d* if the sum, over all vertices, of the absolute value of the difference of the out-degree and the in-degree is equal to $2d$. The deficiency can vary between 0 (for an Eulerian graph) and $|E|$.

A vertex v is said to be *balanced* if $id(v) = od(v)$. A vertex is said to be *partially balanced* (in the partial graph), if $pid(v) = pod(v)$.

A vertex v is a *sink* if $id(v) > od(v)$. A vertex v is a *partial sink* if $pid(v) > pod(v)$. We say the sink is *discovered* (to be a sink) when its partial in-degree exceeds its out-degree. If more edges are later discovered into v, then v remains a partial sink, and its partial deficiency $pid(v) - pod(v)$ increases.

A vertex v is a *source* if $id(v) < od(v)$. A vertex v is an *partial source* if $pid(v) < pod(v)$. Since the partial in-degree can increase over time, a partial source may cease to be a partial source when it becomes partially balanced. It may later even become a partial sink.

4.2 Properties of Deficiency-d Graphs

Recall that during a *walk-based* algorithm whenever the explorer starts traversing unexplored edges she must continue to do so until she gets stuck.

Lemma 4 *A partial sink is a sink if the graph is explored by a walk-based algorithm.*

Proof: If v is a partial sink, then $pid(v) > pod(v)$. This means that the explorer came into v by traversing $pid(v)$ incoming edges, but left v on only $pod(v)$ outgoing edges. Thus, at least one outgoing edge of v is traversed twice. In a walk-based algorithm this can only happen if all outgoing edges are explored, and the explorer was not able to take an unexplored outgoing edge. We have $pod(v) = od(v)$, and therefore,

$$id(v) \geq pid(v) > pod(v) = od(v).$$

Since $id(v) > od(v)$, v is a sink. □

Lemma 5 *In a walk-based algorithm, a walk from an unfinished vertex u in G either ends in u (loops) or ends at either a sink of G or a partial source of G_p.*

Proof: Assume that the explorer takes a walk from vertex u and gets stuck in vertex v. If $v = u$, the walk created a loop. We show that if $v \neq u$, v is either a sink or a partial source. We distinguish the two cases that $pid(v) > pod(v)$ and $pid(v) \leq pod(v)$ after the walk.

If $pid(v) > pod(v)$ after the walk, then v is a partial sink after the walk. Since the graph is explored by a walk-based strategy, it follows from Lemma 4 that the partial sink v is a sink. Thus, the explorer got stuck in a sink of G.

If $pid(v) \leq pod(v)$ after the walk, then the partial in-degree of v must have been strictly less than the partial out-degree of v before the walk. This follows from the following observations. The partial in- and out-degree of v increases by one whenever v is traversed during the walk.

Vertex v must have had at least one unexplored incoming edge e before the walk started. After traversing this edge e during the walk, the explorer is stuck in v. Thus, the partial in-degree after the walk is increased by one more than the partial out-degree is increased. See Figure 4-1.

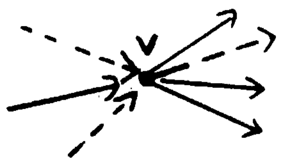


Figure 4-1: After taking a walk, the explorer is stuck in a vertex v whose partial in-degree is smaller than its partial out-degree. Edges that are explored before the walk are illustrated with a straight line, and edges that are explored after the walk are illustrated with a dotted line.

If $pid(v) < pod(v)$ before the walk, then vertex v must have been a partial source before the walk. □

Lemma 6 *A graph of deficiency d has at most d sinks and d sources.*

Proof: First let us note that the set of vertices of any directed graph v exclusively consists of balanced vertices, sinks, and sources. Also,

$$\sum_{\text{sources } v \in V} (od(v) - id(v)) = \sum_{\text{sinks } v \in V} (id(v) - od(v)) \tag{4.1}$$

for any directed graph v. For the balanced vertices of a directed graph, we have

$$\sum_{\text{balanced } v \in V} (od(v) - id(v)) = 0 \tag{4.2}$$

For a graph with deficiency d, we have

$$d = \frac{1}{2} \sum_{v \in V} |id(v) - od(v)|, \tag{4.3}$$

by definition. This means that

$$\sum_{\text{sinks } v \in V} (id(v) - od(v)) + \sum_{\text{sources } v \in V} (od(v) - id(v)) = 2d. \tag{4.4}$$

It follows from (4.1) that

$$\sum_{\text{sources } v \in V} (od(v) - id(v)) = d, \tag{4.5}$$

and

$$\sum_{\text{sinks } v \in V} (id(v) - od(v)) = d. \tag{4.6}$$

Since at most d sources can attribute to the sum in equation (4.5), and at most d sinks to the sum in equation (4.6), it follows that a graph with deficiency d has at most d sinks and d sources. □

Lemma 7 *The partial graph of a graph of deficiency d has at most d partial sources and d partial sinks during a walk-based exploration.*

Proof: It follows from Lemma 4 that in a partial graph that is explored by a walk-based algorithm every partial sink is a sink. Since there are at most d sinks in the graph, there are at most d partial sinks in the partial graph.

We know from equation (4.1) that

$$\sum_{\text{partial sources } v \in V_p} (pod(v) - pid(v)) = \sum_{\text{partial sinks } v \in V_p} (pid(v) - pod(v)) \tag{4.7}$$

for the partial graph. Lemma 4 tells us that

$$pod(v) = od(v), \tag{4.8}$$

if v is a partial sink. This gives us the following relation between the partial graph and the unknown graph for a vertex v that is a partial sink:

$$\begin{aligned} pid(v) - pod(v) &= pid(v) - od(v) && (4.9) \\ &\leq id(v) - od(v) && (4.10) \end{aligned}$$

Equation (4.9) follows from (4.8), and inequality (4.10) follows from the fact that $pid(v) \leq id(v)$. We conclude that

$$\begin{aligned}
\sum_{\textit{partial sources } v \in V_p} (pod(v) - pid(v)) &= \sum_{\textit{partial sinks } v \in V_p} (pid(v) - pod(v)) \\
&\leq \sum_{\textit{sinks } v \in V_p} (pid(v) - pod(v)) && (4.11) \\
&\leq \sum_{\textit{sinks } v \in V} (id(v) - od(v)) && (4.12) \\
&= d && (4.13)
\end{aligned}$$

Equation (4.11) follows from Lemma 4, equation (4.12) from inequality (4.10), and equation (4.13) from equation (4.6). Since at most d partial sources can contribute to

$$\sum_{\textit{partial sources } v \in V_p} (pod(v) - pid(v)),$$

it follows that the partial graph has at most d partial sources during a walk-based exploration. □

In this chapter we have shown properties of deficiency-d graphs. In particular, we described properties of the partial graph of a graph that is explored by a walk-based strategy. We use these properties in the correctness proofs in the following chapter.

Chapter 5

An Algorithm for Deficiency–One Graphs

In this chapter we discuss an algorithm that explores an unknown graph G of deficiency zero or one. The algorithm is a combination of two algorithms due to Deng and Papadimitriou [DP90]. The analysis of this algorithm is based on the analysis that Deng and Papadimitriou give for their algorithms.

A graph with deficiency one has one source and one sink. Given the start vertex s, the algorithm DEFICIENCY-ONE explores the whole graph without any prior information on whether the deficiency is zero or one. Each edge in the graph is traversed at most four times during the exploration.

5.1 Outline of the Algorithm

The DEFICIENCY-ONE algorithm solves the problem of exploring an unknown graph of deficiency one or zero, given a start vertex s. DEFICIENCY-ONE is directly based on the EULERIAN-EXPLORATION algorithm. Like the Eulerian algorithm, DEFICIENCY-ONE uses the technique of taking walks, and working on the created paths. In a graph of deficiency one, there is only one source and one sink (as shown in Lemma 6). During a walk-based exploration, the explorer loops, or gets stuck at the sink or at the partial source (Lemma 5). Whenever the explorer gets stuck, she must relocate (see the definition in section 2.2) and traverse a finished path until she reaches an unfinished vertex from where she starts to take a new walk.

We show that after an initial phase, the paths of the partial graph that need to be finished and the paths that are traversed for relocation are connected in a certain configuration throughout the exploration. Once the partial graph reaches this structure, we call the procedure FINISH, and explore the whole graph by calling FINISH recursively.

To get to the point where we can use FINISH, we must deal with several initial cases of the partial graph. DEFICIENCY-ONE determines if the graph has deficiency zero or one. If the deficiency is one, DEFICIENCY-ONE distinguishes the cases where the partial source *ps* is reachable from the current node, and where it is not. If *ps* is not reachable, we call the procedure REACH-PS that chooses the paths to be worked on with the goal to make the partial source reachable as soon as possible. Once *ps* is reachable, procedure FINISH is called.

5.2 The Finish Procedures

The procedure FINISH is called once the partial graph contains a certain structure that we call a FINISH-structure. FINISH continues the exploration by working on the unfinished paths of the FINISH-structure. If the explorer gets stuck in a partial source, the partial graph contains a FINISH-structure again and FINISH can be called recursively.

The FINISH-structure consists of five paths A, B, C, D, and E (see Fig. 5-1(a)). The explorer is at vertex $D[0]$. The paths A, B, C, and D form a cycle, i.e., the last vertex on path A is the first vertex of path B, the last vertex on B is the first vertex on C and so on. Path E is connected to the cycle by its first vertex: $E[0]$ is the same vertex as $B[0]$. Since there are two paths starting at the same vertex a, where $a = B[0] = E[0]$, a is the partial source of the partial graph. Paths A and C are finished. To stress this property of A and C, we use the notation $\tilde{A}$ and $\tilde{C}$ to indicate that A and C are finished paths. The FINISH-structure of the partial graph is illustrated in Figure 5-1(a). We illustrate the finished paths with a zigzag line.

We call a FINISH-structure *reduced* if its cycle only consists of two paths $\tilde{A}$ and B where the last vertex on B is $\tilde{A}[0]$ and the last element on $\tilde{A}$ is $B[0]$. Note that the reduced FINISH-

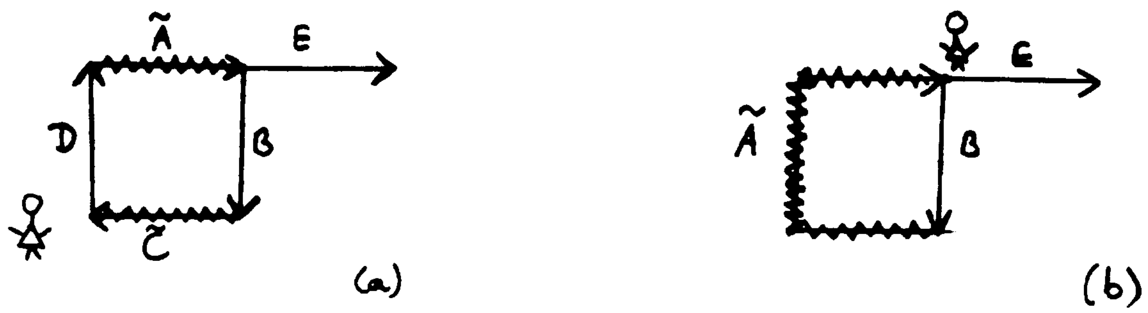


Figure 5-1: (a) The FINISH-structure of a partial graph. (b) The reduced FINISH-structure.

structure is a FINISH-structure where paths $\tilde{C}$ and D are empty. The reduced FINISH-structure of the partial graph is illustrated in Figure 5-1(b).

First we define a procedure FINISH which works on a partial graph that contains a FINISH-structure. Later we define a procedure FINISH-R which works on a partial graph that has a reduced FINISH-structure.

The input of the procedure FINISH consists of the graph G, the partial graph G_p, and five paths $\tilde{A}, B, \tilde{C}, D$, and E that describe the unfinished paths in G_p and how they are connected. FINISH calls procedure WORK-ON on path D which means that the explorer tries to finish path D first. Depending on the outcome of the work on D, either procedure FINISH-R, or procedure FINISH is called recursively. FINISH is called recursively on an input of five paths that describe a FINISH-structure.

```
FINISH($G, G_p, \tilde{A}, B, \tilde{C}, D, E$)            ▷ Input illustrated in Fig. 5-1(a)
 1  ($new$,, $Newpath$ $i$,) ← WORK-ON($G, G_p, D$)
 2  if $new$ is false                             ▷ Path $\tilde{D}$ is finished
 3     then move to $B[0]$ along path $\tilde{A}$        ▷ See Fig. 5-3(a)
 4          $G_p$ ← FINISH-R($G, G_p$,                ▷ See Fig. 5-3(b)
 5                          $\tilde{C}\tilde{D}\tilde{A}$,             ▷ new $\tilde{A}$ is concatenation of paths $\tilde{C}, \tilde{D}$, and $\tilde{A}$
 6                          $B$,               ▷ new $B$ is old $B$
 7                          $E$)               ▷ new $E$ is old $E$
 8     else ▷ stuck at $B[0]$ while taking a walk from $D[i]$, see Fig. 5-2
 9          $G_p$ ← FINISH($G, G_p$,
10                          $\tilde{C}\tilde{D}[..i]$,          ▷ new $\tilde{A} = \tilde{C}$ and finished prefix $\tilde{D}[..i]$ of $D$
11                          $D[i..]$,          ▷ new $B$ = unfinished prefix $D[..i]$
12                          $\tilde{A}$,               ▷ new $\tilde{C}$ = old $\tilde{A}$
13                          $B$,               ▷ new $D$ = old $B$
14                          $Newpath$ $E$)     ▷ new $E$ = concatenation of $Newpath$ and $E$
15  return $G_p$
```

When the procedure FINISH is called, the explorer starts out working on path D. Lemma 5 says that the explorer either loops, or gets stuck at a sink or source when working on a path. Since the sink is found before FINISH is called, the explorer cannot get stuck at a sink during the execution of FINISH.

Thus, the explorer can either get stuck at the partial source $B[0]$ while taking a walk from some vertex $D[i]$ on path D, or finish D. Figure 5-2 illustrates the situation in which the explorer gets stuck at node $B[0]$ while working on D.

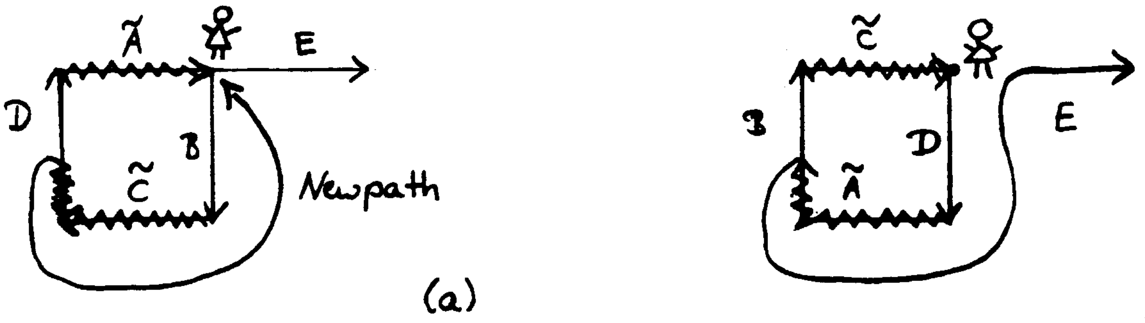


Figure 5-2: (a) Partial graph after getting stuck at vertex $B[0]$ while working on D in line 8 of FINISH. (b) FINISH-structure of recursive call of FINISH in lines 9 to 14 of FINISH.

Procedure WORK-ON returns the path *Newpath* which is the new unfinished path in the partial graph that is created during the walk from $D[i]$. The explorer is at the old partial source $B[0]$.

In the following, we show that the partial graph now has a FINISH-structure again, so that FINISH can be called recursively.

Since vertex $D[i]$ is the new partial source in the graph, it takes on the function of vertex $B[0]$ in the new FINISH-structure that is input to the recursive call of FINISH in lines 9–14.

When the explorer gets stuck at $B[0]$ while working on path D, D is not finished completely. The portion $D[i..] = < D[i], \ldots, D[l_D] >$, where $D[l_D]$ is the end of path D and $D[l_D] = A[0]$, is unfinished. It takes on the function of path B in the following recursive call of FINISH (line 11).

The finished prefix $D[..i] = < D[0], \ldots, D[i] >$ of D is appended to path $\tilde{C}$ and takes on the function of path $\tilde{A}$ in the recursive call of FINISH (line 10).

Path B takes on the function of path D in the recursive call of FINISH (line 13); it is the path that the explorer will work on next.

The concatenation of paths E and *Newpath* takes on the function of path E in the recursive call of FINISH (line 14). The renaming of paths described above is illustrated in Figure 5-2(b). Notice that paths $\tilde{A}, B, \tilde{C}, D$ and E form a FINISH-structure again.

Since the number of unexplored edges in the graph reduces every time we call FINISH recursively in line 9 and work on path D, D is eventually finished at some point during the execution of FINISH. See Figure 5-3(a). Then the explorer moves along path $\tilde{A}$, and procedure FINISH-R is called recursively.

FINISH-R takes a reduced FINISH-structure as an input. In lines 4, 5, 6, and 7 of the procedure FINISH, the input for the call of FINISH-R is defined. The concatenation of paths $\tilde{C}$, $\tilde{D}$, and $\tilde{A}$ takes on the function of path $\tilde{A}$ in FINISH-R. Paths B and E remain the same. We illustrate how the partial graph in FINISH looks like when FINISH-R is called in Figure 5-3(b).

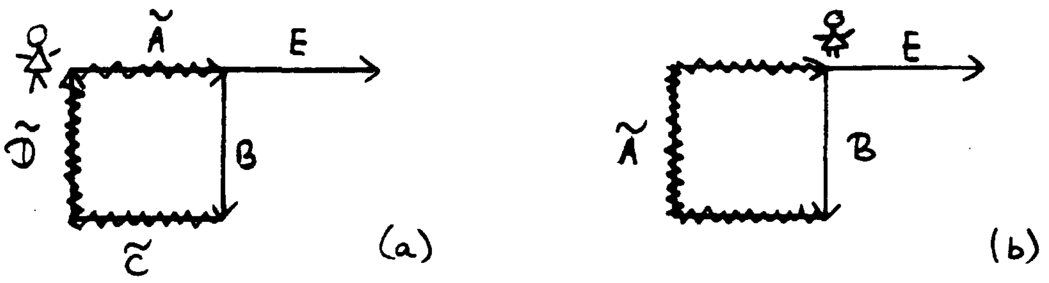


Figure 5-3: (a) Partial graph after path D is finished in line 2 of FINISH. (b) FINISH-structure of recursive call of FINISH-R in lines 4 to 7 of FINISH.

In the following, we define the procedure FINISH-R which is a simpler version of the procedure FINISH. The inputs of the procedure FINISH-R are the graph G, the partial graph G_p, and three paths $\tilde{A}, B$, and E that form the reduced FINISH-structure of a partial graph.

```
FINISH-R($G, G_p, \tilde{A}, B, E$)                ▷ Input is illustrated in Fig. 5-1(b)
 1  ($new$, $Newpath$ , $i$) ← WORK-ON($G, G_p, B$)
 2  if $new$ is $false$                           ▷ Path $B$ is finished. See Fig. 5-5
                                                  ▷ Path $E$ is the only unfinished path in the graph.
 3      then move to $E[0]$ along $\tilde{A}$
 4           ($new$, $Newpath$ , $i$) ← WORK-ON($G, G_p, E$)
 5           if $new$ is $false$                  ▷ Path $E$ is finished.
 6              then return $G_p$                 ▷ The graph is explored.
 7              else ▷ stuck at $E[0]$ while taking a walk from $E[i]$. See Fig. 5-6
 8                   move to $E[i]$ along $E$
 9                   $G_p$ ← FINISH-R($G, G_p$,       ▷ See Fig. 5-6
10                                   $\tilde{E}[..i]$,    ▷ new $\tilde{A}$ is finished prefix of $E$
11                                   $Newpath$,     ▷ new $B$ = $Newpath$
12                                   $E[i..]$)      ▷ new $E$ is suffix of old $E$
13      else ▷ stuck at $B[0]$ while taking a walk from $B[i]$. See Fig. 5-4
14           move to $B[i]$ along $B$
15           $G_p$ ← FINISH-R($G, G_p$,             ▷ See Fig. 5-4
16                           $\tilde{A}\tilde{B}[..i]$,          ▷ new $\tilde{A}$ is old $\tilde{A}$ and finished prefix $\tilde{B}[..i]$,
17                           $B[i..]$,           ▷ new $B$ is unfinished suffix of $B$
18                           $Newpath$ $E$)      ▷ new $E$ = $Newpath$ and $E$
```

When the procedure FINISH-R is called, the explorer starts out working on path B. We know by Lemma 5 that the explorer either loops, or gets stuck at the partial source when working on path B.

Thus, the explorer can either get stuck at the partial source $B[0]$ while taking a walk from some vertex $B[i]$ on path B, or finish B. We illustrate the situation in which the explorer gets stuck at node $B[0]$ while working on B in Figure 5-4(a).

In the following, we show that the partial graph now contains a reduced FINISH-structure again, so that FINISH-R can be called recursively.

Since vertex $B[i]$ is the new partial source in the graph, it takes on the function of vertex $B[0]$ in the new reduced FINISH-structure that is input of a recursive call of FINISH-R.

When the explorer gets stuck at $B[0]$ while working on path B, B is not finished completely. The portion $< B[i], \ldots, B[l_B] >$, where $B[l_B]$ is the end of path B and $B[l_B] = A[0]$, is unfinished. It takes on the function of path B in the following recursive call of FINISH.

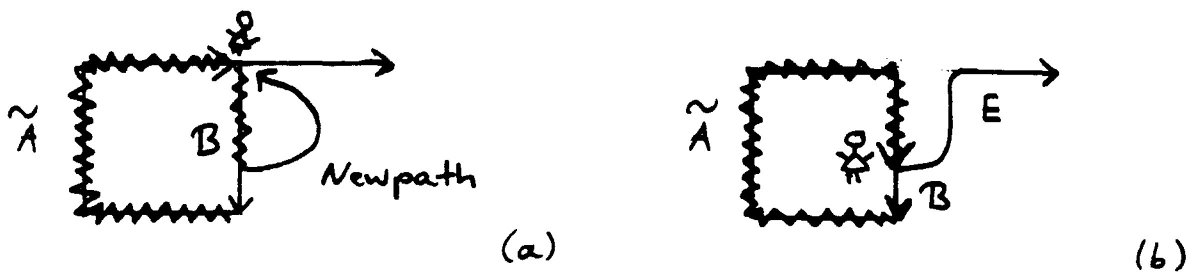


Figure 5-4: Partial graph after getting stuck at vertex $B[0]$ while taking a walk from $B[i]$ in line 13 of FINISH-R. (b) FINISH-structure of recursive call of FINISH-R in lines 15 to 18 of FINISH-R.

The finished portion $< B[0], \ldots, B[i] >$ of B is appended to path $\tilde{A}$ and takes on the function of path A in the recursive call of FINISH.

Path E is appended to the created path *Newpath*; the new E is $< B[i], \ldots, E[0], \ldots, E[l_E] >$, where l_E is the length of path E. This path takes on the function of path E in the recursive call of FINISH. The renaming of paths described above is illustrated in Figure 5-4(b). Notice that paths A, B, and E form a reduced FINISH-structure again.

Path B is eventually finished at some point during the execution of FINISH-R. See Figure 5-5(a). Then the explorer moves to $E[0]$ along path $\tilde{A}$, and starts working on path E. See Figure 5-5(b). Paths $\tilde{A}$ and B are not needed for relocation anymore. We circle the portion of the FINISH-structure that can be discarded (from the FINISH-structure, but not from G_p) with a dotted line in our figures.

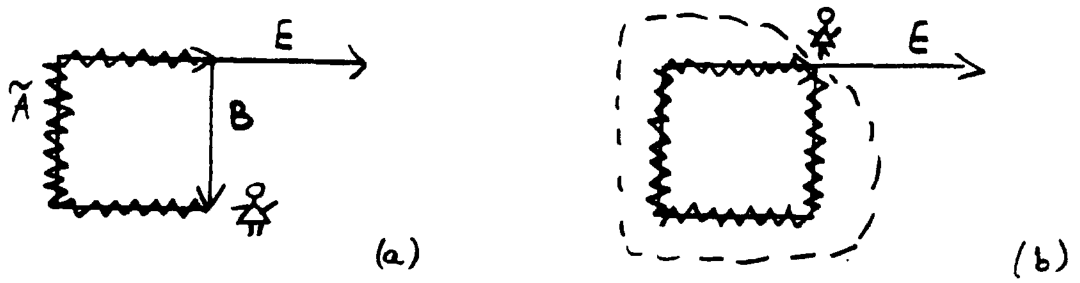


Figure 5-5: Partial graph (a) after path $\tilde{B}$ is finished in line 2 of FINISH-R and (b) when procedure WORK-ON is called in line 4.

The explorer can either get stuck at the partial source $E[0]$ while taking a walk from some vertex $E[i]$ on path E, or finish E. If the explorer gets stuck in $E[0]$ (see Fig. 5-6(a)), it traverses E until she reaches $E[i]$ and calls procedure FINISH-R recursively. The first formal parameter $\tilde{A}$ of FINISH-R which is input to the recursive call in line 9 is the finished portion

$\tilde{E}[..i] = < E[0], \ldots, E[i] >$ of path E. The second parameter B is the path *Newpath* that has been created after the explorer took a walk from $E[i]$. The third parameter E is the unfinished suffix of path E. The partial graph that is input to the recursive call of procedure FINISH-R is illustrated in Figure 5-6(b).

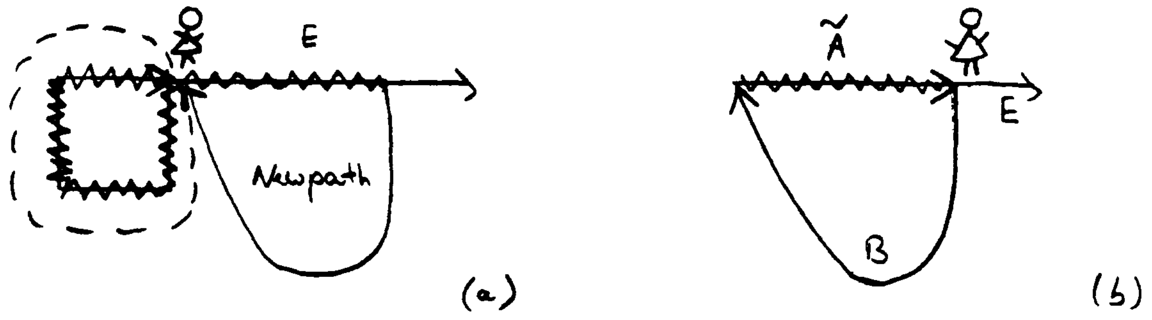


Figure 5-6: (a) Partial graph after getting stuck at vertex $E[0]$ while taking a walk from vertex $E[i]$ (line 7 of FINISH-R). (b) FINISH-structure of recursive call of FINISH-R in lines 9 to 12 of FINISH-R.

If the explorer finishes path E, every path that is part of the FINISH-structure is finished. Then procedure FINISH-R returns the partial graph G_p to its caller in line 6. We show below that the returned partial graph G_p is the same as graph G.

Assume that during the exploration of a graph of deficiency-one we have the following situation. The sink v of the graph has been found, i.e., $v \in G_p$, and the partial graph contains a FINISH-structure. The edges on the finished paths $\tilde{A}$ and $\tilde{C}$ of the FINISH-structure have been traversed at most twice and the edges on the unfinished paths B, D, and E at most once. Every edge in the partial graph that is not on a path that is part of the FINISH-structure has been traversed at most four times and is on a finished path. Every unexplored edge has not been traversed at all. We call this situation the ***input assumptions of procedure*** FINISH.

Now assume that we have the following situation during the exploration of a graph of deficiency-one. The sink v of the graph has been found, i.e., $v \in G_p$, and the graph contains a reduced FINISH-structure. The edges on $\tilde{A}$ of the reduced FINISH-structure have been traversed at most three times and the edges on the unfinished paths B and E at most once. Every edge in the partial graph that is not on a path that is part of the reduced FINISH-structure has been traversed at most four times and is on a finished path. We call this situation the *input assumptions of procedure* FINISH-R.

In the following lemma, we show that if procedure FINISH is called on a partial graph for which the input assumptions of FINISH hold, then the input assumptions of FINISH hold for any recursive calls of FINISH. We also show that FINISH-R is called correctly on a partial graph for which the input assumptions of procedure FINISH-R are satisfied.

Lemma 8 *Assume that procedure* FINISH *is called on a partial graph for which the input assumptions of* FINISH *hold. Then procedure* FINISH *continues to explore the graph and calls either procedure* FINISH *on a partial graph for which the input assumptions of procedure* FINISH *hold, or procedure* FINISH-R *on a partial graph for which the input assumptions of procedure* FINISH-R *hold.*

Proof: We have argued above that the work on path D in line 1 of FINISH ends in only two cases of the partial graph, and we have shown that for both cases the partial graph contains a (reduced) FINISH-structure, so that either FINISH is called recursively, or FINISH-R is called.

Every relocation in FINISH in line 3 is done along the loop that consists of paths $\tilde{A}$, $B, \tilde{C}$, and D. Therefore, every relocation is possible.

It remains to show that the trace of every edge in the partial graph satisfies the input assumptions for the recursive call of FINISH and the call of FINISH-R.

Every edge in the graph is traversed once when it is explored. So every edge that is explored during the execution of the procedure FINISH is traversed once when it is explored. FINISH calls procedure WORK-ON on the unfinished path D of the FINISH-structure, so the explorer traverses edges on D a second time. Any edge in the partial graph, whether it is explored before or during the execution of FINISH, is traversed additional times only when the explorer must relocate. Every edges that is part of the FINISH-structure of the partial graph is also part of the new FINISH-structure of the partial graph in when procedure FINISH is called recursively in lines 9-14 of FINISH.

Since the input assumptions were satisfied when FINISH is called, i.e., the edges on D have been traversed at most once, the trace on the edges of the prefix of D is two after line 8. Since D is appended to a finished path (with edge traces = 2) in the FINISH-structure of the recursive call of procedure FINISH in lines 9-14, FINISH has an input that satisfies the *input assumptions* of FINISH.

In the following, we consider the case that the explorer relocates during the execution of the procedure FINISH. There is only one relocation in procedure FINISH which is in line 3.

Consider the partial graph after relocation in line 3 of FINISH. The edges on path $\tilde{A}$ are traversed for the third time since the exploration has started. See Figure 5-7. For every path in the Figure, we illustrate the trace of the edges on the path.

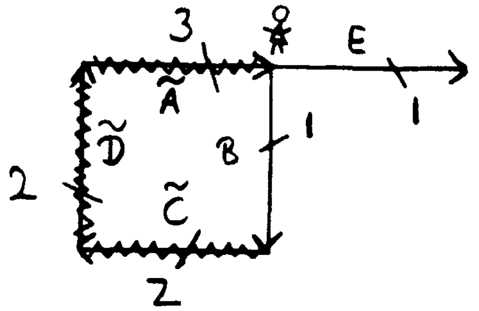


Figure 5-7: Partial graph after relocating in line 3 of FINISH.

Line 3 of FINISH can only be executed once during the exploration of the graph, because once path D is finished, we do not call FINISH recursively again. The partial graph after relocation in line 3 contains a reduced FINISH-structure where the trace of the edges on paths B and E is one, on paths C and D is two, and on path A is three. The concatenation $\tilde{C}\tilde{D}\tilde{A}$ is input path A of FINISH-R. Edges on this path have been traversed at most three times and every edge that has been part of the FINISH-structure when FINISH was called is now part of the reduced FINISH-structure. Therefore, the input assumptions of FINISH-R are satisfied. □

Lemma 9 *Assume that procedure* FINISH-R *is called on a partial graph for which the input assumptions of* FINISH-R *hold. Then procedure* FINISH-R *continues to explore the graph and calls procedure* FINISH-R *on a partial graph for which the input assumptions of procedure* FINISH-R *hold.*

Proof: We have argued above that the work on path B in line 1 of FINISH-R ends in only two cases of the partial graph, and we have shown that for each case the partial graph contains a reduced FINISH-structure, so that either case FINISH-R is called recursively. In both cases, the necessary relocation is possible, because it is done along the loop that consists of paths $\tilde{A}$ and B. We have shown above in which cases the work on B and on E in FINISH-R ends, and that the recursive calls of FINISH-R in lines 9–12 and lines 15-18 have an input which contains a reduced FINISH-structure.

It remains to show that the trace of every edge in the partial graph satisfies the input assumptions of the recursive calls of FINISH-R. As in Lemma 8, we argue that every edge that is explored during the execution of the procedure FINISH-R has been traversed once, and every edge on path B that is finished during the execution of the procedure FINISH-R has been traversed twice. Any edge in the partial graph, whether it is explored before or during the execution of FINISH-R, is traversed additional times only when the explorer relocates. Every edge in the partial graph that has been traversed four times already (and is therefore not part of the reduced FINISH-structure when FINISH-R is called initially), is not traversed during relocation. This observation follows from the fact that only edges that are part of the reduced FINISH-structure of the partial graph are traversed during relocation.

Relocation happens in lines 3, 8, and 14 of FINISH-R. We illustrate the partial graph before and after the line 14 is executed in Figure 5-8.

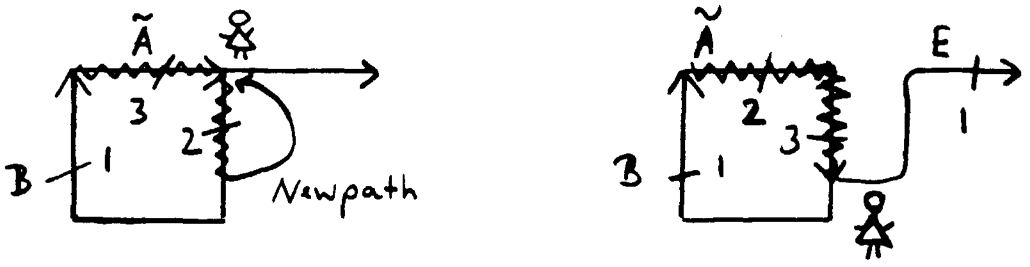


Figure 5-8: Partial graph before and after line 14 of FINISH-R is executed.

We call the vertex that is $B[0]$ the first time that line 13 of FINISH-R is executed a_1, the vertex that is $B[0]$ the second time line 13 is executed a_2, and the vertex that is $B[0]$ the jth time line 13 is executed a_j. Notice that $a_1, a_2, \ldots, a_j, \ldots$ are all vertices on the original path B. Relocations after getting stuck at a_1 when taking a walk from a_2 involves traversing edges on $a_1 \rightsquigarrow a_2$ for the third time. In general, relocations after getting stuck at partial source a_i involves traversing $a_i \rightsquigarrow a_{i+1}$. Since $a_i \rightsquigarrow a_{i+1}$ and $a_{i+1} \rightsquigarrow a_{i+2}$ are different portions of the original path B, no edge on B is traversed more than once for relocation in line 14 of FINISH-R.

We call the property that the partial source moves closer to $A[0]$ every time the explorer gets stuck in some partial source a_i and only traverses $a_i \rightsquigarrow a_{i+1}$ to relocate *"the partial source moves cheaply down a path."* See Figure 5-9.

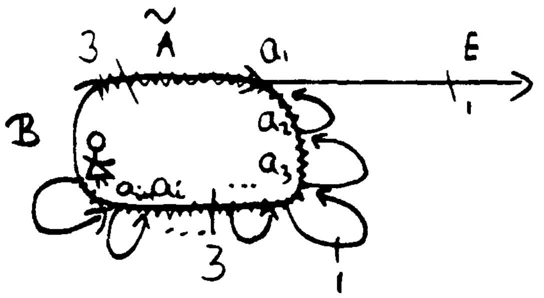


Figure 5-9: Partial graph with a partial source that is "moving cheaply down the path."

Once the explorer reaches $A[0]$, the cycle that consists of $\tilde{A}$ and $\tilde{B}$ is finished. The relocation that is needed to get to $E[0]$ in line 3 of FINISH-R involves traversing the edges on the cycle from $A[0]$ to $E[0]$ again. Thus, the edges on the cycle have been traversed at most four times when the work on path E is started. See Figure 5-10(a).

Notice that any further call of FINISH-R means that relocation is needed along path E, but not along cycle $B[0] \rightsquigarrow B[0]$. Indeed, the cycle is no longer part of the reduced FINISH-structure of the partial graph after line 3 FINISH-R, as illustrated in Figure 5-10(b). Thus, the input assumptions for the recursive call of FINISH-R in lines 9–12 are satisfied.

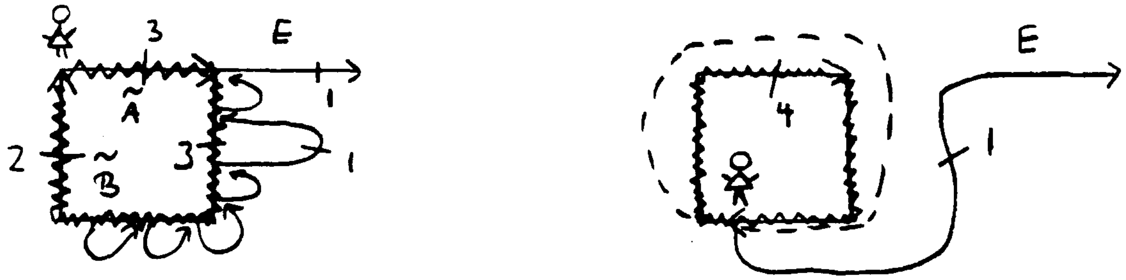


Figure 5-10: Partial graph before and after relocation in line 3 of FINISH-R.

Work on E may stop when the explorer gets stuck at $E[0]$ while taking a walk from some vertex $E[i]$ on E. In this situation the property that the partial source moves cheaply down path $E[0] \rightsquigarrow E[i]$ holds and the edges on $E[0] \rightsquigarrow E[i] \rightsquigarrow E[0]$ are traversed at most four times before they are not part of the FINISH-structure anymore. See Figure 5-11. □

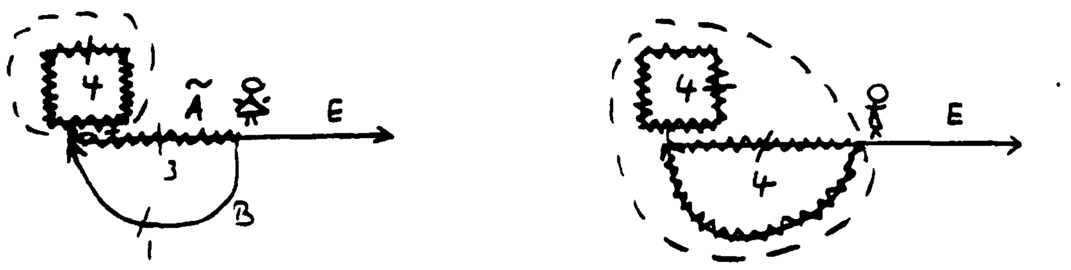


Figure 5-11: Partial graph after relocating in line 8 of FINISH-R for the first and second time.

Lemma 10 *Procedure* FINISH-R *returns the explored graph.*

Proof: Procedure FINISH-R returns the partial graph in line 6 after the work on path E in line 4 is finished. We argued above that paths $\tilde{A}$ and $\tilde{B}$ are finished already. It follows from the input assumptions of procedure FINISH-R that every path in the graph that is not contained in the FINISH-structure of the input to FINISH-R is finished.

Assume that G_p is not explored completely. Then there is either an unfinished vertex x on some path G_p or there is an undiscovered vertex w in G ($v \in V - V_p$). Since all the paths in G_p are finished, the assumption that there exists an unfinished vertex x immediately leads to a contradiction.

In the following, we show that the assumption that there is an undiscovered vertex w in G also leads to a contradiction. We use the strong connectivity of G that to argue that w is connected to the rest of the graph. There exists a path from a discovered vertex s to w. (There is at least one discovered vertex in a partial graph - the start vertex.) Therefore, there exists an edge that leads from a discovered vertex a to an undiscovered vertex b on this path. Vertex a is on some path in G_p. Note that a is unfinished, because edge (a, b) is unexplored. Since the paths in G_p are all finished, we have a contradiction. □

Procedure FINISH returns the partial graph that is returned by FINISH-R. Therefore, procedure FINISH also returns a graph that is explored entirely.

5.3 The Reach-ps Procedure

Before the exploration of a graph of deficiency one leads to a partial graph that contains a FINISH-structure, the partial graph may have the property that the explorer cannot reach the partial source by traversing edges in the partial graph. We mentioned in Chapter 2 that the partial graph may not be strongly connected during the exploration, although the graph is strongly connected. We say that the partial source is not *reachable* for the explorer.

The procedure REACH-PS tells the explorer how to work on the reachable part of the partial graph until the partial source is also reachable.

When REACH-PS is called, the partial graph is assumed to have one of the structures illustrated in Figure 5-12(a) or (b). The explorer is at sink v when REACH-PS is called.

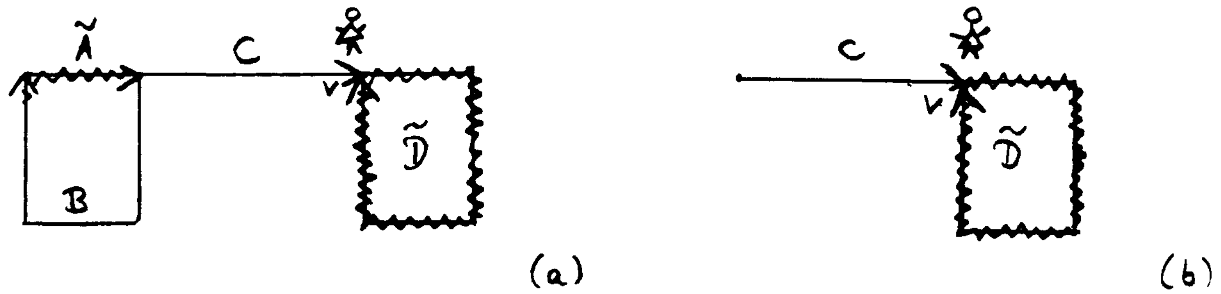


Figure 5-12: The partial graph that is an input to procedure REACH-PS is of one of these forms: (a) G_p has four nonempty paths, (b) paths A and B are empty.

The input of the procedure REACH-PS consists of the graph G, the partial graph G_p, and four paths $\tilde{A}, B, C$, and $\tilde{D}$ that describe the unfinished paths in G_p and how they are connected. Paths $\tilde{A}$ and B may be empty, as in Figure 5-12(b).

The procedure REACH-PS consists of two parts. The first part is a repeat-loop that is used to force the reachability of the partial source by working on the reachable parts of path C until the partial source $C[0]$ is reachable.

The second part of REACH-PS determines how to continue the exploration of the graph, once the partial source is reachable. We distinguish the case that the explorer gets stuck at $C[0]$, and the case that a vertex on path B is reached during a walk while working on C. We show that in each case the partial graph has a FINISH-structure, so the procedure FINISH is called to finish the exploration of the graph.

```
REACH-PS($G, G_p, \tilde{A}, B, C, \tilde{D}$)    ▷ See Fig. 5-12 for input $G_p$
1  $i \leftarrow l_C$                              ▷ i=length of $C$
2   repeat $j \leftarrow i$
3           move to vertex $C[i]$ where $i$ is the smallest index such that $C[i]$ is reachable from $C[j]$
4           $S \leftarrow$ SUBPATH($C, i, j$)
5           ($new$,$Newpath$,$k$) $\leftarrow$ WORK-ON($G, G_p, S$)
6     until $C[0]$ is reachable
   ▷ See Fig. 5-14 for current partial graph $G_p$
7  $E \leftarrow A[0]$                    ▷ Create a new empty path $E$
8  if stuck at $C[0]$ while taking a walk from $S[k]$
      ▷ $new$ = $true$. See Fig 5-14(a).
9     then $G_p \leftarrow$ FINISH($G, G_p$,          ▷ See Fig. 5-15
10                          $S[..k]$             ▷ new $\tilde{A}$ = prefix of $S$
11                          $Newpath$ $B$        ▷ new $B$ = old $B$ appended to $Newpath$
12                          $\tilde{A}$                ▷ new $\tilde{C}$ is old $\tilde{A}$
13                          $C[..i]$,            ▷ new $D$ = suffix of $C$
14                          $S[k..]$)            ▷ new $\tilde{E}$ = unfinished suffix of $S$
15    else  ▷ Vertex $B[m]$ on $B$ is reachable, $<C[i], \ldots, C[j]>$ is finished. See Fig 5-14(b).
16          move to $B[m]$
17          $G_p \leftarrow$ FINISH($G, G_p$,
18                          $\tilde{A}$,               ▷ new $\tilde{A}$ = old $\tilde{A}$
19                          $B[..m]$,            ▷ new $B$ = prefix $<B[0], \ldots B[m]>$
20                          $<B[m]>$,            ▷ new $C$ is empty path with vertex $B[m]$
21                          $B[m..]$,            ▷ new $D$ = suffix of $B$
22                          $C[..i]$)            ▷ new $\tilde{E}$ = prefix old $<C[0], \ldots C[i]>$
23  return $G_p$
```

The procedure REACH-PS works as follows. In lines 1–6 the partial source is made reachable by working on different portions $C[i] \rightsquigarrow C[j]$ of path C. In lines 7–14 the input paths for the FINISH-procedure are defined. For simplicity, we first assume that the repeat-loop is executed only once.

Note that the vertices on paths $\tilde{A}$ and B are distinct from the vertices on paths C and $\tilde{D}$, because otherwise the explorer could reach the partial source $C[0]$ and REACH-PS would not have been called. However, there is at least one vertex on $\tilde{D}$ (other than $D[0]$) that is also on path C, because otherwise there would not be a connection from path D to the rest of the graph G. We know that this cannot occur, because the graph to be explored is strongly connected. Thus, there is a vertex $C[i]$ on C that the explorer can reach from $D[0]$. Among the reachable nodes $C[i], C[i'], C[i''], \ldots$ on C, we pick in line 3 the vertex $C[i]$ with the smallest index (i.e., $i < i' < i'' \ldots$).

While the explorer works on the portion $S = C[i] \rightsquigarrow D[0]$ of path C in line 5 of REACH-PS, she may get stuck at $C[0]$, and the repeat-loop is left. See Figure 5-13(a).

The explorer may also finish the portion $S = C[i] \rightsquigarrow D[0]$ of path C, and may have spliced a walk through a vertex $B[m]$ on path B into S. Since $C[0]$ is the same vertex as $B[0]$, there is a path from $D[0]$ to $C[0]$, and the partial source $C[0]$ is reachable and the repeat-loop is terminated. See Figure 5-13(b).

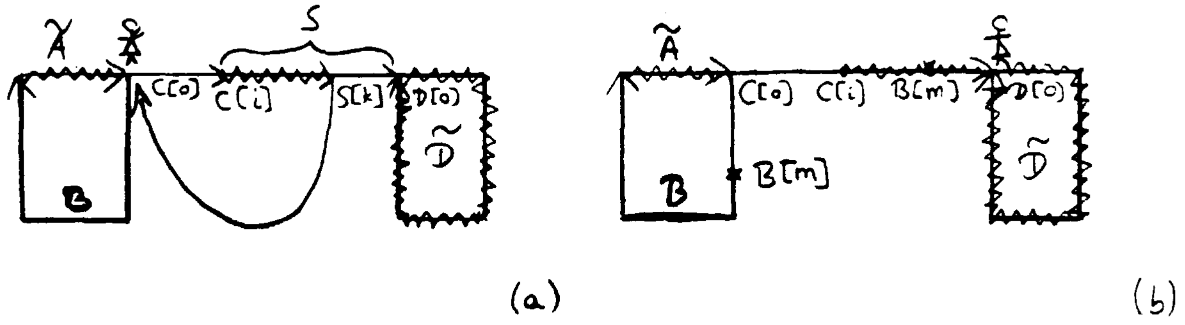


Figure 5-13: Partial graph after the repeat-loop is executed only once before it is left: (a) stuck at $C[0]$ while working on portion $C[i] \rightsquigarrow D[0]$ of path C; (b) $C[0]$ is reachable after $C[i] \rightsquigarrow D[0]$ is finished

Now we consider the more general case that the repeat-loop is executed more than once. Finishing path $C[i] \rightsquigarrow D[0]$ may result in the case that the partial source $C[0]$ is not reachable. Then index j is updated with index i, and a new $C[i]$ on the portion $C[0] \rightsquigarrow C[j]$ of path C is found whose index i is smaller than j. We use the same argument as above to show that vertex $C[i]$ exists (or the explorer gets to path B): Since G is strongly connected, path $C[j] \rightsquigarrow D[0]$ and path D are connected to the rest of the partial graph. Thus, there must be a vertex v on $C[j] \rightsquigarrow D[0]$ that connects this portion of C to some unfinished part of G_p. This vertex v cannot be on path $\tilde{A}$, because $\tilde{A}$ is finished. If v is on B, the loop is left. If the loop is executed more than once, v must be on some unfinished part of C. If there are several such vertices v, the vertex $C[i]$ with the smallest index i on path C is chosen.

Working on $C[i] \rightsquigarrow C[j]$ may result in making $C[0]$ reachable or repeating the loop. The loop is left eventually, because the unfinished portion of C is finite, so that the index i becomes smaller every time the loop is executed, until $C[0]$ is reachable. See Figure 5-14.

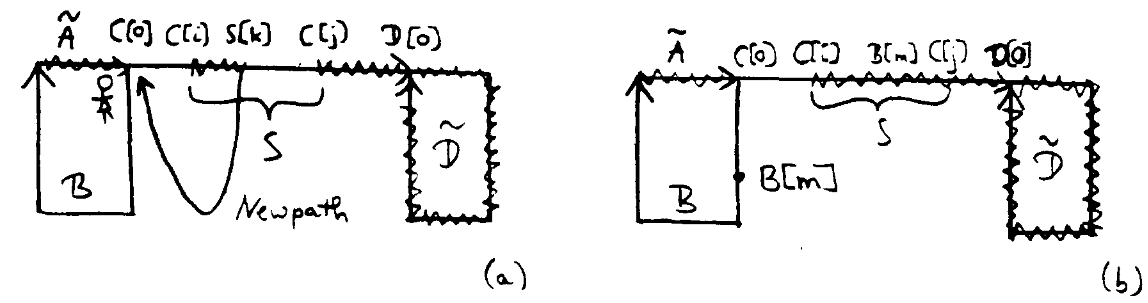


Figure 5-14: Partial graph after the repeat-loop is executed several times before it is left: (a) stuck at $C[0]$ while working on portion $C[i] \rightsquigarrow C[j]$ of path C;(b) $C[0]$ is reachable after $C[i] \rightsquigarrow C[j]$ is finished

In the following, we show that the partial graph has a FINISH-structure when the repeat-loop is left. Assume that the condition in line 8 of REACH-PS is true, the explorer is at the partial source $C[0]$, and the partial graph looks like the graph in Figure 5-14(a). Note that we can distinguish eight different paths, four of them unfinished. The finished paths are $\tilde{A}$, $\tilde{D}$, the prefix $C[i] \rightsquigarrow S[k]$ of S, and path $C[j] \rightsquigarrow C[l_C]$, where l_C is the length of path C and $C[l_C] = D[0]$. We discard paths $\tilde{D}$ and $C[j] \rightsquigarrow C[l_C]$. The unfinished paths of the partial graph are B, the prefix $C[0] \rightsquigarrow C[i]$ of C, the suffix $S[k] \rightsquigarrow C[j]$ of S, and the path *Newpath* from $S[k]$ to $C[0]$ that was created during the last walk. See Figure 5-15(a).

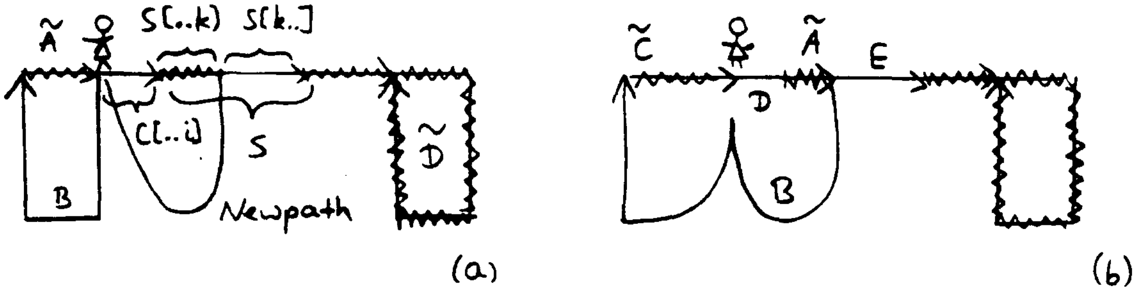


Figure 5-15: (a) Partial graph after getting stuck at $C[0]$ (line 9 of REACH-PS). (b) FINISH-structure of the input paths to procedure FINISH in lines 10–14 of REACH-PS.

We concatenate the *Newpath* with path B to obtain a new path that we call B. Now we have a cycle of four paths: $B, \tilde{A}$, $C[0] \rightsquigarrow C[i]$, and $C[i] = S[0] \rightsquigarrow S[k]$. When we rename path $\tilde{A}$ to be $\tilde{C}$, $< C[0], \ldots C[i] >$ to be D, and $< S[0] \ldots S[k] >$ to be $\tilde{A}$, we see that the partial graph contains a FINISH-structure consisting of this cycle and path $< S[k] \ldots C[j] >$ as path E. The explorer is at vertex $D[0]$. See Figure 5-15(b). Thus, the procedure FINISH in called on a valid input and can continue to explore the graph.

If the partial graph on which procedure REACH is called looks like the graph in Figure 5-16(a) where paths A and B are empty, the input paths of procedure FINISH in line 9–14 of REACH-PS form a valid FINISH-structure in which path $\tilde{C}$ is empty. See Figure 5-16(b).

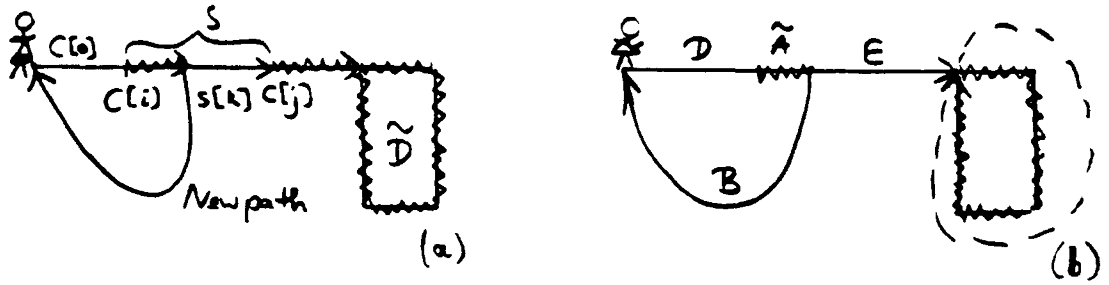


Figure 5-16: (a) Partial graph with empty paths $\tilde{A}$ and B in line 8 of REACH-PS. (b) FINISH-structure of G_p that is input to procedure FINISH in lines 9–14.

Now assume that the condition in line 8 of REACH-PS is false. Path S is finished and the explorer is at $C[j]$. There is some vertex $B[m]$ on path S that the explorer can move to and finish exploring the graph by calling FINISH in line 17 of REACH-PS. We show that the input paths to procedure FINISH in lines 17–22 of REACH-PS form a proper FINISH-structure.

Vertex $B[m]$ takes on the function of $D[0]$ in the FINISH-structure, so the suffix $B[m..]$ of B takes on the function of path D. Input path $\tilde{C}$ is defined to be an empty path containing vertex $B[m]$. The prefix $B[..m]$ of B takes on the function of B, and the subpath $C[0] \rightsquigarrow C[i]$ of C takes on the function of E.

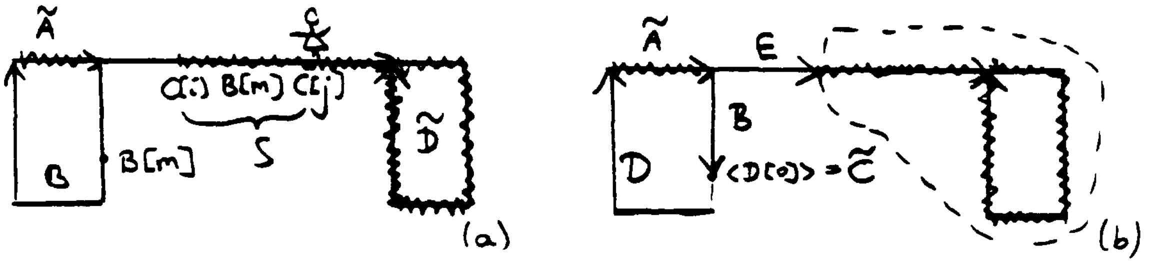


Figure 5-17: (a) Partial graph with finished path S in line 16 of REACH-PS.(b) FINISH-structure that is input to procedure FINISH in lines 17–22 of REACH-PS.

If the partial graph on which procedure REACH is called has empty paths $\tilde{A}$ and B i.e.,$A = B = < C[0] >$, (see Figure 5-18(a)), then the input paths to procedure FINISH in lines 17–22 of REACH-PS form a valid FINISH-structure in which $\tilde{A}, B, \tilde{C}$ and D are empty paths. The vertex reachable vertex $B[m]$ that is found in line 15 of REACH-PS is $B[0] = C[0]$. See Figure 5-18(b).

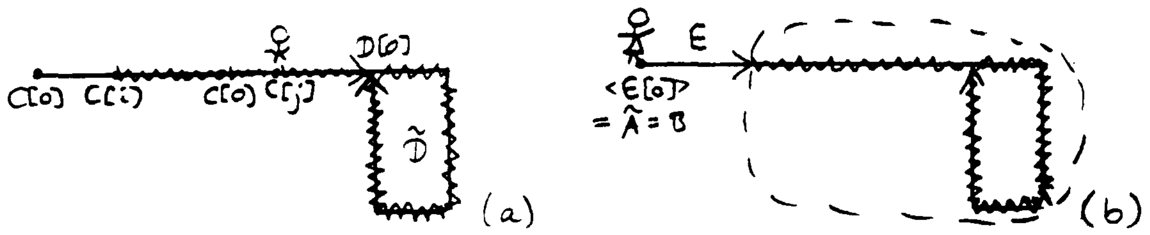


Figure 5-18: (a) Partial graph with empty paths $\tilde{A}$ and B in line 16 of Reach-ps. (b) Finish-structure of input to Finish in lines 17–22 of Reach-ps.

Assume that during the exploration of a graph of deficiency-one we have the following situation. The sink v of the graph has been found, i.e., $v \in G_p$, and the graph has the proper input structure to procedure Reach-ps as illustrated in Figure 5-12. The edges on the finished paths $\tilde{A}$ and $\tilde{D}$ have been traversed at most twice and the edges on the unfinished paths B and C at most once. Every edge in G that is on a path that is not part of the input structure of Reach-ps has not been explored, and therefore not traversed at all. We call this situation the *input assumptions* of procedure Reach-ps.

In the following lemma, we show that if procedure Reach-ps is called on a partial graph for which the input assumptions of Reach-ps hold, then the input assumptions of Finish hold for any calls of Finish during the execution of Reach-ps.

Lemma 11 *Assume that procedure* Reach-ps *is called on a partial graph for which the input assumptions of* Reach-ps *hold. Then procedure* Reach-ps *continues to explore the graph and calls procedure* Finish *on a partial graph for which the input assumptions of procedure* Finish *hold (as defined in section 5.2).*

Proof: The correctness of procedure Reach-ps follows from the fact that during the execution of the repeat-loop the partial source becomes reachable, and that then the partial graph has indeed a Finish-structure, so that calling the procedure Finish succeeds in exploring the whole graph.

We showed above that there exists a vertex $C[i]$ on the prefix $C[0] \rightsquigarrow C[j]$ of C, where $0 \leq i < j$, that is reachable. Since index j is updated with index i in line 2 of Reach-ps, the number of vertices between $C[0]$ and $C[j]$ on path C decreases every time the repeat-loop is executed. Thus, eventually partial source $C[0]$ becomes reachable. We also showed above that the partial graph contains a FINISH-structure, so that it is proper to call procedure FINISH in lines 9-14 and lines 17-22.

It remains to show that the trace of every edge in the partial graph satisfies the *input assumptions* for the call of FINISH.

Every edge that is explored during the execution of the procedure REACH-PS is traversed once when it is explored. REACH-PS calls procedure WORK-ON on the reachable portion of path C, so the explorer traverses edges on C a second time. Any edge in the partial graph, whether it is explored before or during the execution of REACH-PS, is traversed additional times only when the explorer must relocate.

In the following, we discuss how often the edges in G_p are traversed for relocation during the execution of the procedure REACH-PS. There are two relocations in procedure REACH-PS which are in lines 3 and 16.

When the explorer moves from $D[0]$ to $C[i]$ during the first execution of the repeat-loop, she traverses edges on $\tilde{D}$ for the third time (relocation in line 3 of REACH-PS).

If the explorer gets stuck at the partial source $C[0]$ while taking a walk from vertex $S[k]$ on $S = C[i] \leadsto D[0]$, the partial graph contains a FINISH-structure where $S[k]$ is the last vertex on path E. Thus, in any recursive call of FINISH, the explorer need not traverse path $\tilde{D}$ anymore.

If the explorer finishes path $S = C[i] \leadsto D[0]$, index i is renamed j, and the explorer moves to a new vertex $C[i]$ on path C that has a smaller index than the former index i. This relocation involves traversing the prefix of path $\tilde{D}$ to get to the former $C[i]$ which is now $C[j]$, and from there along S to the new $C[i]$. Some of the edges on $\tilde{D}$ are traversed for the fourth time, and some of the edges of S are traversed for the third time. See Figure 5-19.

If the explorer gets stuck at the partial source $C[0]$ while taking a walk from vertex $S[k]$ on $S = C[i] \leadsto C[j]$ the second time line 5 of REACH-PS is executed, again the partial graph has a FINISH-structure where $S[k]$ is the last vertex on path E. Thus, in any recursive call of FINISH, the explorer need not traverse path $\tilde{D}$ or any edge on C that has been traversed three times already.

If the explorer does not get stuck at the partial source after line 5 of REACH-PS is executed for the second time, and the partial source is still not reachable, a new vertex $C[i]$ (third $C[i]$) is determined in line 2 of REACH-PS, and the explorer moves to it in line 3 of REACH-PS. This

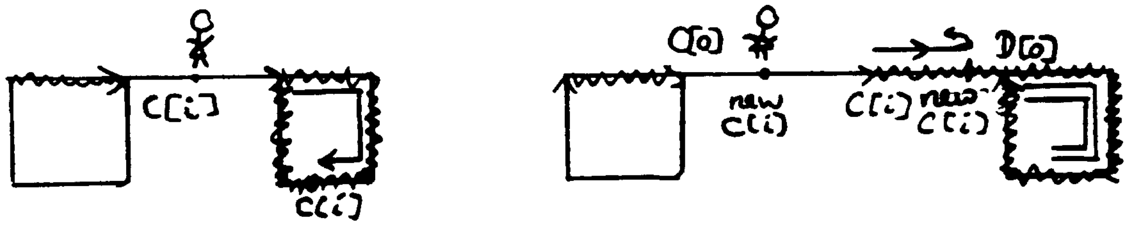


Figure 5-19: Traversals of edges on path C and $\tilde{D}$ after line 3 of REACH-PS is called for the (a) first and (b) second time.

relocation does not involve any traversal of path $\tilde{D}$ anymore. The explorer follows the path *first* $C[i] \rightsquigarrow D[0]$ (=path S the first time the loop was executed) to the *second* $C[i]$, from there she follows path *second* $C[i] \rightsquigarrow$ *first* $C[i]$ until she reaches the new (third) $C[i]$. The prefix of path *first* $C[i] \rightsquigarrow D[0]$ is traversed for the fourth time, the prefix of path *second* $C[i] \rightsquigarrow$ *first* $C[i]$ for the third time. See Figure 5-20.

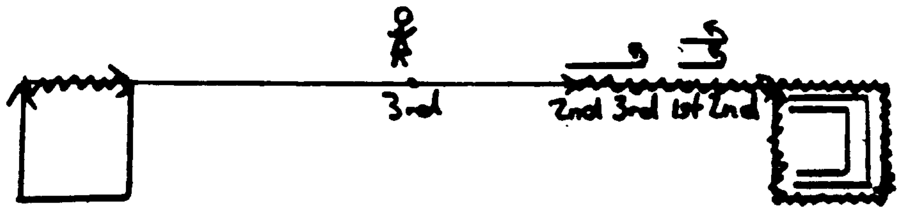


Figure 5-20: Traversals of edges on path C after the second relocation in line 3 of REACH-PS.

In general, any edge on C or that is explored from a vertex on C is traversed at most four times during the repeat-loop: once when the edge is explored, once when the portion of C on which the edge lies is finished, and twice for relocation to a portion of C that is "closer" to the partial source $C[0]$. See Figure 5-21.

If a vertex $B[m]$ on S becomes reachable after the repeat-loop is executed several times, the relocation to $B[m]$ involves the same portions of C that would have been traversed if $B[m]$ were a new $C[i]$. Therefore, no edge on C has been traversed more than four times after relocation in line 16 of REACH-PS. See Figure 5-22.

If the explorer gets stuck in $C[0]$ while taking a walk from some vertex $S[k]$ on path S, the

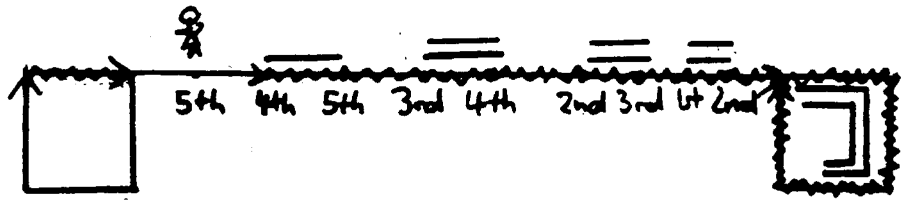


Figure 5-21: Traversals of edges on path C after k relocations in line 3 of REACH-PS.

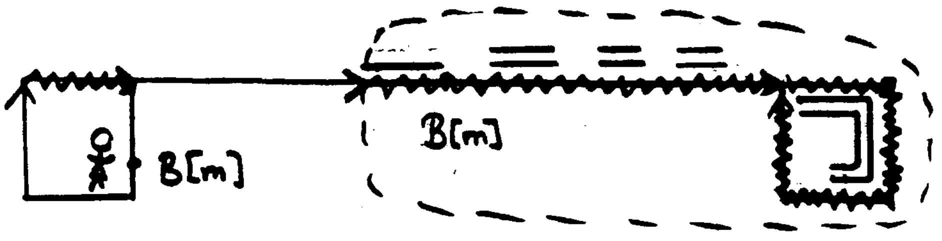


Figure 5-22: Traversals of edges on path C after a vertex $B[m]$ is reachable. The repeat-loop has been traversed several times.

edges on $S[k] \rightsquigarrow D[0]$, and $\tilde{D}$ are not part of the FINISH-structure of the partial graph, so they are not traversed in recursive calls of FINISH. See Figure 5-23.

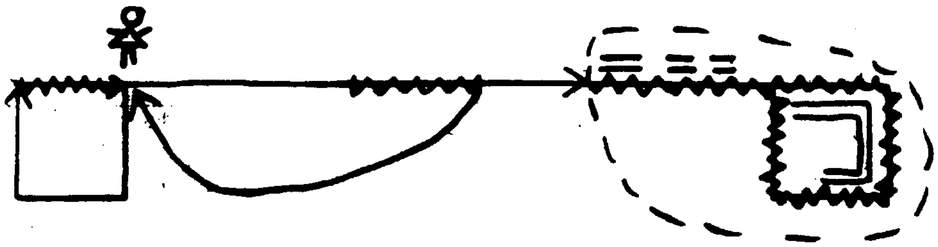


Figure 5-23: Traversals of edges on path C after getting stuck at $C[0]$. The repeat-loop has been traversed several times.

Before the repeat-loop terminates, the edges on paths $\tilde{A}$ and B have not been traversed at all during the execution of REACH-PS, because they were not reachable. The second part of the procedure (lines 7-23) renames paths, the explorer does not move along paths $\tilde{A}$ and B, so no edges on $\tilde{A}$ and B are traversed. Therefore, edges on paths $\tilde{A}$ and B are traversed at most twice, when FINISH is called in lines 9-14 and 17-22. We have shown above that the paths whose edges have been traversed four times are not part of the input to the procedure FINISH. Thus, the trace of every edge in the partial graph satisfies the input assumptions for the call of FINISH. As we have shown above, any relocation during FINISH does not involve edges that are not part of the input FINISH-structure, so the edges on the discarded path $\tilde{D}$ are not traversed anymore.□

Lemma 12 *Procedure* REACH-PS *returns the explored graph.*

Proof: Procedure REACH-PS returns the partial graph in line 23 after procedure FINISH returns. We argued above that the input assumptions to FINISH are satisfied, when FINISH is called in lines 9–14 and lines 17–22. It follows by lemma 10 that the graph returned by REACH-PS is explored. □

5.4 The Deficiency–One Algorithm

After having introduced the procedures FINISH and REACH-PS we define the algorithm DEFICIENCY-ONE that explores a graph of deficiency zero or one by calling the basic operations WALK and WORK-ON and the procedures FINISH and REACH-PS. DEFICIENCY-ONE takes an input graph G and a start vertex s and returns the partial graph G_p after G is explored. The partial graph G_p that is returned by the DEFICIENCY-ONE algorithm is equal to the graph G.

In the DEFICIENCY-ONE algorithm, the explorer starts exploring the graph from vertex s until she either finishes exploring the whole graph if the graph has deficiency zero, or until she gets stuck in the sink of a deficiency one graph. In the following implementation, the procedures SINK-CASE or LOOP-CASE are called depending on the structure of the partial graph of a deficiency-one graph.

```
DEFICIENCY-ONE($G, s$)
 1  $P \leftarrow$ WALK($G, G_p, s$)
 2  if path $P$ is a loop
 3     then ($new$, $Newpath$, $i$) $\leftarrow$ WORK-ON($G, G_p, P$)
 4           if $new$ = false          ▷ no new path is created, $P$ is finished; deficiency-zero.
 5             then return $G_p$        ▷ Graph is explored.
 6           elseif stuck at sink $P[m]$ on path $P$
               ▷ Graph is a deficiency-one graph; see Fig. 5-25(a)
 7             then $G_p \leftarrow$ FINISH($G, G_p$,      ▷ See Fig. 5-25(b)
 8                              $P[..i]$,     ▷ new $\tilde{A}$ = prefix of $P$
 9                              $P[i..m]$,    ▷ new $B$ = portion $<P[i], \ldots P[m]>$ of $P$
10                              $<P[m]>$,     ▷ new $\tilde{C}$ is empty path with vertex $P[m]$
11                              $P[m..]$,     ▷ new $D$ = suffix of $P$
12                              $Newpath$)    ▷ new $\tilde{E}$ = $Newpath$
13             else ▷ stuck at sink that is not on path $P$; see Figure 5-24(b).
14                  $G_p \leftarrow$ LOOP-CASE($G, G_p, P, Newpath, i$)
15     else ▷ Path $P$ is not a loop; explorer stuck at a vertex $v$, $v \neq P[0]$; see Figure 5-24(a).
16           determine smallest index $i$ such that $v = P[i]$
17           $G_p \leftarrow$ SINK-CASE($G, G_p, P, i$)
18  return $G_p$
```

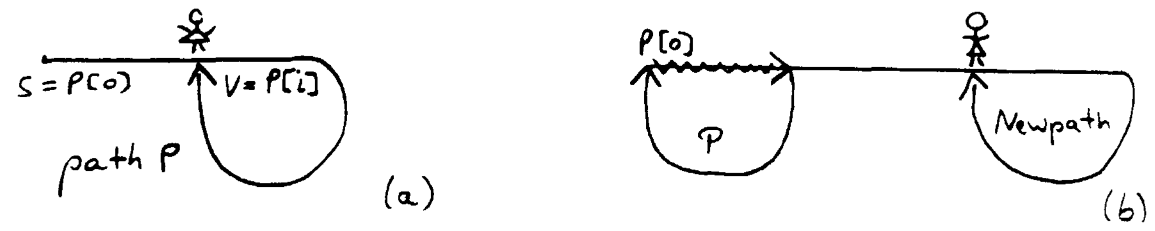


Figure 5-24: Partial Graph after the sink is found: (a) P is not a loop and SINK-CASE is called in line 9 of DEFICIENCY-ONE, (b) path P is a loop and work on P ends in sink v on path *Newpath*

The DEFICIENCY-ONE algorithm works as follows. The explorer starts with a walk from start vertex s in line 1. If the first walk from the start vertex s in line 1 is a loop, the graph is either a deficiency-zero or a deficiency-one graph. If the graph has deficiency zero, which means that it is a Eulerian graph, working on the path created by the walk is sufficient to finish exploring the whole graph (lines 3– 5). If the graph has deficiency one, working on the path created by the walk ends in getting stuck in the sink v of the graph. This is illustrated in Figure 5-24(b).

Line 6 of the DEFICIENCY-ONE algorithm checks if the explorer got stuck in some vertex

$P[m]$ on path P or in a vertex not on path P, but only on path *Newpath*. The partial graphs that may result after work on path P in a deficiency-one graph are illustrated in Figures 5-24(b) and 5-25(a). Figure 5-25(a) shows a partial graph in which the walk on vertex $P[i]$ ended in some node $P[m]$ on path P. Figure 5-24(b) shows a partial graph in which the walk on vertex $P[i]$ ended in some node on path *Newpath*. In the case that the sink is on path P, the DEFICIENCY-ONE algorithm determines index m in line 6 and calls procedure FINISH.

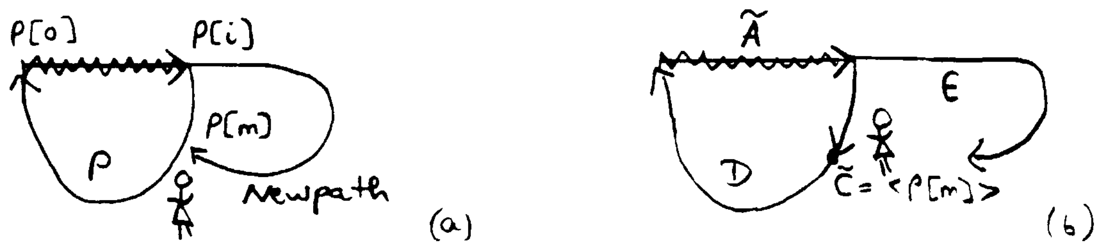


Figure 5-25: (a) Partial graph of line 6 of procedure DEFICIENCY-ONE: the walk from vertex $P[i]$ ended in $P[m]$ on path P (b) Partial Graph that is input to procedure FINISH in line 7–12 of procedure DEFICIENCY-ONE.

In the following, we show that the partial graph contains a FINISH-structure, and the *input assumptions* for FINISH (as defined in Section 5.2) are satisfied. Note that the loop that is formed by path P can be interpreted as the cycle $\tilde{A}, B, \tilde{C}$ and D in a FINISH-structure, where $P[..i] =< P[0], \ldots, P[i] >$ takes on the function of path $\tilde{A}$, $P[i..m] = P[i], \ldots, P[m] >$ the function of path B, $< P[m] >$ the function of path $\tilde{C}$, and $P[m..] =< P[m], \ldots, P[0] >$ the function of path D. Since *Newpath* is attached to $P[i] = B[0]$, it is a valid path E in the FINISH-structure. The explorer is at $P[m] = D[0]$. Thus, the partial graph contains a FINISH-structure on which procedure FINISH is called in lines 7–12 of the DEFICIENCY-ONE algorithm. The input to FINISH in lines 7–12 is illustrated in Figure 5-25(b). During the walk in line 1 of DEFICIENCY-ONE the edges on P are traversed once; during the work on P in line 3 of DEFICIENCY-ONE the edges on P are traversed again, and the edges on *Newpath* are traversed once. Thus, the trace of edges on path A of the FINISH-structure is two and the trace of edges on B, D and E is one. Path C does not contain an edge. It follows that the *input assumptions* for FINISH are satisfied when procedure FINISH is called in lines 7-12 of DEFICIENCY-ONE.

The procedure LOOP-CASE is called in line 14 of procedure DEFICIENCY-ONE to continue exploring a deficiency-one graph if the explorer is not stuck on path P, but on path *Newpath*. We first define procedure LOOP-CASE before we continue describing algorithm DEFICIENCY-ONE and the procedure SINK-CASE.

The inputs of the procedure LOOP-CASE are the graph G, the partial graph G_p, the cyclic path P, the path *Newpath* that is created during a walk from a node $P[i]$ on P, and the index i. The trace of the edges on the suffix $P[i..]$ of P and on *Newpath* is one and the trace of the edges on the prefix $P[..i]$ of P is two. Procedure LOOP-CASE works on the suffix of *Newpath* and calls either FINISH or REACH-PS depending on the outcome of this work. The input of procedure LOOP-CASE is illustrated in Figure 5-26(a). In the following implementation of procedure LOOP-CASE, the prefix of *Newpath* that ends with the sink of the graph is called path S, and the suffix of *Newpath* is called path T. See Figure 5-26(b).

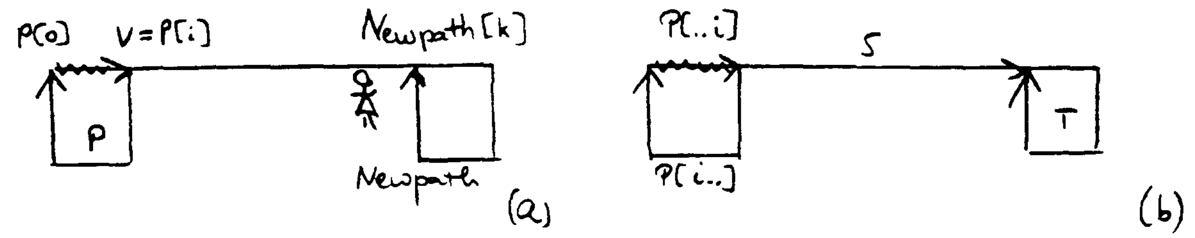


Figure 5-26: (a) Partial Graph that is input to procedure LOOP-CASE (b) Partial graph before work on path T starts in line 4 of procedure LOOP-CASE.

```
LOOP-CASE($G, G_p, P$, $Newpath$, $i$)
    ▷ Explorer is stuck at sink $v$. See Figure 5-26(a)
 1  Determine smallest index $k$ such that $v = Newpath[k]$
 2  $S \leftarrow Newpath[..k]$              ▷ $S$ is prefix of $Newpath$
 3  $T \leftarrow Newpath[k..]$              ▷ $T$ is suffix of $Newpath$
 4  $(new, W, q) \leftarrow$ WORK-ON($G, G_p, T$)
 5  if stuck at $P[i]$ while taking a walk from vertex $T[q]$
    ▷ See Fig. 5-27.
 6     then $G_p \leftarrow$ FINISH($G, G_p$,
 7                     $T[..q]$,     ▷ new $\tilde{A}$ = prefix of $P$
 8                     $W\ P[i..]$,  ▷ new $B$ = suffix of $P$ is appended to path $W$
 9                     $P[..i]$,     ▷ new $C$ = prefix of $P$
10                     $S$,          ▷ new $D$ = path $S$
11                     $T[q..]$)     ▷ new $E$ = suffix of $T$
12     else ▷ cycle $T$ is finished
13           if some vertex $P[r]$ is now on $\tilde{T}$
             ▷ See Fig. 5-28(a).
14              then move to $P[r]$ along $T$
15                   $G_p \leftarrow$ FINISH($G, G_p$,     ▷ See Fig. 5-28(b)
16                            $P[..i]$,     ▷ new $\tilde{A}$ = prefix of $P$
17                            $P[i..r]$,    ▷ new $B$ = portion $<P[i], \ldots P[r]>$ of $P$
18                            $<P[r]>$,     ▷ new $C$ is empty path with vertex $P[r]$
19                            $P[r..]$,     ▷ new $D$ = suffix of $P$
20                            $S$)          ▷ new $E = S$
21              else ▷ partial source still not reachable, see Fig. 5-29
22                   $G_p \leftarrow$ REACH-PS($G, G_p$,
23                            $P[..i]$,   ▷ new $\tilde{A}$ = prefix of $P$
24                            $P[i..]$,   ▷ new $B$ = suffix of $P$
25                            $S$,        ▷ new $C = S$
26                            $\tilde{T}$)        ▷ new $\tilde{D} = \tilde{T}$
27  return $G_p$
```

Procedure LOOP-CASE works as follows. In line 4 of LOOP-CASE, the explorer starts working on cycle T. If the explorer gets stuck at the partial source $P[i]$ while taking a walk from a vertex $T[q]$ on T, procedure FINISH is called. The input to procedure FINISH is illustrated in Figure 5-27. The input paths to procedure FINISH are the finished prefix $T[..q]$ of T as input parameter $\tilde{A}$, prefix $P[..i]$ as input parameter $\tilde{C}$, unfinished path S as parameter D, and the unfinished suffix $T[q..]$ of T as parameter E. Parameter B is the new created walk W concatenated with the unfinished portion $P[i] \rightsquigarrow P[0]$ of P. The explorer is at vertex $P[i] = S[0]$ which takes on the function of $D[0]$ in the FINISH-structure of the partial graph. The edges on $P[i..]$, $E = T[q..]$ and $D = S$ are not traversed in lines 1-5 of LOOP-CASE, so the trace of these edges is still one as at the time of the call of LOOP-CASE. The edges on $\tilde{A} = P[..i]$ are also not traversed during the execution of LOOP-CASE, so the trace is still two as at the time of the call of LOOP-CASE. *Newpath*$[k..] = T$ has been traversed once when the explorer starts to work on it in line 4 of LOOP-CASE. After the explorer is stuck at $P[i]$, the trace of the edges on prefix $T[..q]$ is two. Thus, the *input assumptions* of procedure FINISH are satisfied when FINISH is called in lines 6-11.

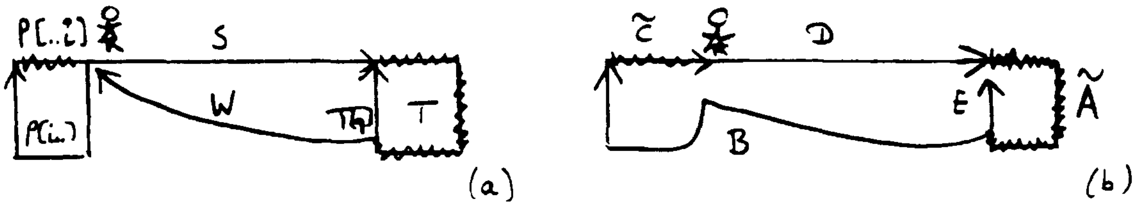


Figure 5-27: (a) Partial Graph after explorer gets stuck at $P[i]$ while taking a walk from $T[q]$. (b) FINISH-structure of partial graph and trace of edges that satisfy *input assumptions* of procedure FINISH in lines 6–11 of procedure LOOP-CASE.

If the explorer does not get stuck in the partial source while working on path T, she finishes path T, and moves to path P if there is a vertex $P[r]$ that is also on the finished path $\tilde{T}$ (lines 12–14 of LOOP-CASE). See Figure 5-28. Vertex $P[r]$ cannot be on the finished prefix $P[..i]$ of P, because the vertices on $P[..i]$ are finished before the vertices on path T are discovered. The partial graph has a FINISH-structure, in which the finished prefix $P[..i] = < P[0], \dots P[i] >$ of P takes on the function of $\tilde{A}$, the portion $P[i..r] = < P[i], \dots P[r] >$ of P takes on the function of B, $\tilde{C}$ is the empty path at vertex $P[r]$, the unfinished suffix $P[r..] = < P[r], \dots P[0] >$ of P takes on the function of D, and S the function of E. Since the explorer is at $P[r]$, this is a valid input to procedure FINISH that is called in lines 15–20 of procedure LOOP-CASE. The trace of the edges on paths B, D, and E is one and the trace of edges on $\tilde{A}$ is two, since the edges are not traversed during the execution of LOOP-CASE. The trace of the edges on T is two after work in line 4 and three after the relocation in line 14. Path T is not part of the FINISH-structure, so the *input assumptions* of procedure FINISH are satisfied, when it is called in lines 15–20.

If the explorer finishes T and still cannot reach path P, procedure REACH-PS is called in lines 22–26 of LOOP-CASE. The input paths to REACH-PS are the finished prefix $\tilde{A}$ of P, the unfinished suffix B of P, and paths S and T as illustrated in Figure 5-29. The trace of the edges on paths B, and C is one and the trace of edges on $\tilde{A}$ and $\tilde{D}$ is two. Therefore, the *input*

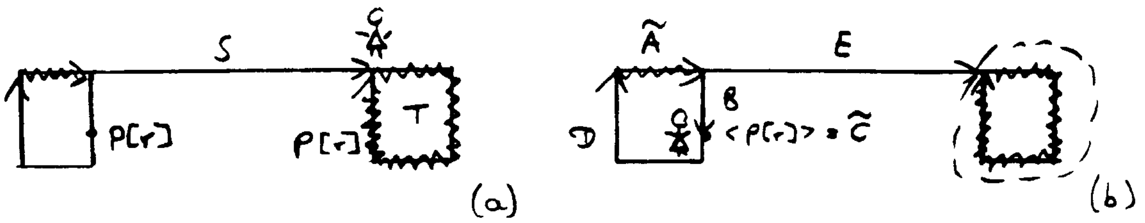


Figure 5-28: (a) Partial Graph after path $\tilde{T}$ is finished and some node $P[r]$ is on $\tilde{T}$ in line 13 of procedure LOOP-CASE. (b) FINISH-structure of partial graph and trace of edges that satisfy *input assumptions* of procedure FINISH in lines 15–20 of procedure LOOP-CASE.

assumptions of procedure REACH-PS as stated in Section 5.3 are satisfied. Procedure REACH-PS starts working on path S until path B becomes reachable.

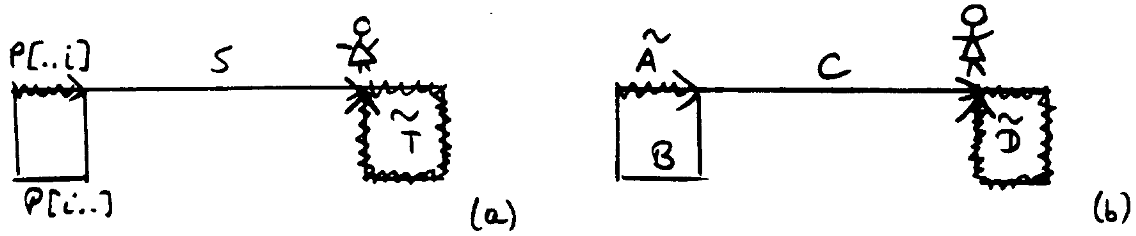


Figure 5-29: (a) Partial Graph before procedure REACH-PS is called in line 22–26 of procedure LOOP-CASE.(b) Partial Graph that is input to procedure REACH-PS in line 22–26 of LOOP-CASE.

Now we continue describing algorithm DEFICIENCY-ONE and define the procedure SINK – CASE that is called in line 17 of the DEFICIENCY-ONE algorithm.

If the first walk from the start vertex s in line 1 is not a loop we know that the explorer got stuck at the sink v as illustrated in Figure 5-24(a). The explorer may have traversed vertex v several times before she got stuck in v. Therefore, vertex v may occur several times on path. We choose to consider the first occurrence of vertex v on path P. This is vertex $P[i]$, where i is the smallest index of the vertices on path P such that $P[i] = v$. Procedure SINK-CASE is called in line 17 of DEFICIENCY-ONE to handle the case where path P ends in the sink.

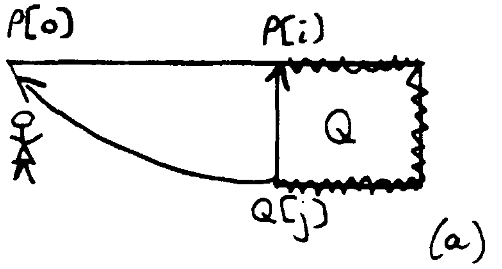
The input to procedure SINK-CASE is the initial walk P, which the explorer takes from the start vertex, and which ends in the sink of the graph. The edges on path P have been traversed once. The input of the procedure SINK-CASE is illustrated in Figure 5-24(a).

```
SINK-CASE($G, G_p, P, i$)        ▷ Path $P$ is not a loop, stuck at $P[i]$; see Fig.5-24(a).
 1  $Q \leftarrow P[i..]$
 2  ($new$, $Newpath$, $j$) $\leftarrow$ WORK-ON($G, G_p, Q$)
 3  if stuck at $P[0]$ while taking a walk from some vertex $Q[j]$
    ▷ See Fig.5-30.
 4     then $G_p \leftarrow$ FINISH($G, G_p,$
 5                            $Q[..j]$,     ▷ new $\tilde{A}$ = prefix of $Q$
 6                            $Newpath$,    ▷ new $B$ = $Newpath$
 7                            $<P[0]>$,     ▷ new $\tilde{C}$ is empty path with vertex $P[0]$
 8                            $P[..i]$,     ▷ new $D$ = prefix of $P$
 9                            $Q[j..]$)     ▷ new $\tilde{E}$ = suffix of $Q$
10     else ▷ path $\tilde{Q}$ is finished, $new$ = false
11          $E \leftarrow P[..i]$
12          $A, B \leftarrow <P[0]>$
13          if $P[0]$ on $Q$
14             then move to $P[0]$       ▷ See Fig. 5-31(a).
15                  $G_p \leftarrow$ FINISH-R($G, G_p, A, B, E$)
16             else ▷ $P[0]$ not reachable from $Q$, see Fig.5-31(b).
17                  $G_p \leftarrow$ REACH-PS($G, G_p, A, B, E, Q$)
18  return $G_p$
```

Procedure SINK-CASE works as follows. The suffix of path P which is a loop from vertex $P[i]$ back to $P[i]$ is called Q in line 1. The explorer starts working on path Q in line 2 of SINK-CASE. The work either ends in getting stuck in the partial source $P[0]$ after which procedure FINISH is called in lines 4–9, or the suffix of P is finished. If the work ended in the partial source $P[0]$, the procedure WORK-ON(G, G_p, Q) in line 2 returns new = **true**. A new walk *Newpath* from some vertex $Q[j]$ to $P[0]$ has been created. This is illustrated in Figure 5-30(a). Procedure FINISH is called on the partial graph in which the finished prefix $<Q[0], \ldots, Q[j]>$ of Q is input parameter $\tilde{A}$, the last walk taken from $Q[j]$ is input parameter B, partial source $P[0]$ is empty input path C, the unfinished prefix $<P[0], \ldots, P[i]>$ is input parameter D, and the unfinished suffix $<Q[j], \ldots, Q[0]>$ of Q is input parameter E. The partial graph has a FINISH-structure as illustrated in Figure 5-30(b). Thus, it is a valid input to procedure FINISH in lines 4–9 of procedure SINK-CASE.

During the work in line 2 of procedure SINK-CASE, the trace of edges on $Q[..j] = \tilde{A}$ is increased by one. Every other path that is part of the FINISH-structure is traversed only once. Thus, the *input assumptions* of procedure FINISH as stated in Section 5.2 are satisfied.

If the work on path Q ends without getting stuck in the partial source $P[0]$, path $\tilde{Q}$, which



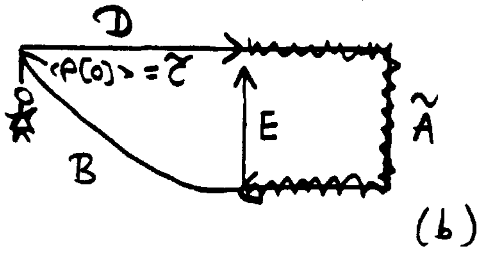
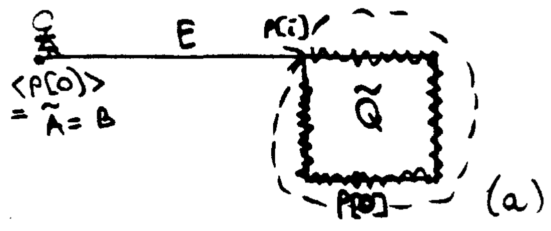


Figure 5-30: ((a) Partial Graph after work on Q ended in the partial source $P[0]$ (line 3 of SINK-CASE). (b) Partial Graph that is input to procedure FINISH in lines 4–9 of SINK-CASE.

is the suffix of P, is finished. Then either the procedure FINISH-R is called if the partial source is reachable, or the procedure REACH-PS is called if the partial source is not reachable. FINISH-R is called on a partial graph that has a FINISH-structure that consists of two empty paths A and B at vertex $P[0]$, and path E which is the unfinished prefix of P (line 11). The *input assumptions* of procedure FINISH-R as stated in Section 5.2 are satisfied, because path $\tilde{Q}$, whose edges are traversed three times, is discarded, and path E is only traversed once. See Figure 5-31(a).

Procedure REACH-PS is called on a partial graph that contains two empty paths A and B, the unfinished path E, and the finished path Q (line 17 of SINK-CASE. After the work in line 2 of SINK-CASE, edges on $\tilde{Q} = \tilde{D}$ have been traversed twice. The edges on $P[..i] = C$ are not traversed during the execution of SINK-CASE. Thus, the *input assumptions* of procedure REACH-PS as stated in Section 5.3 are satisfied. The input to procedure REACH-PS is illustrated in Figure 5-31(b).



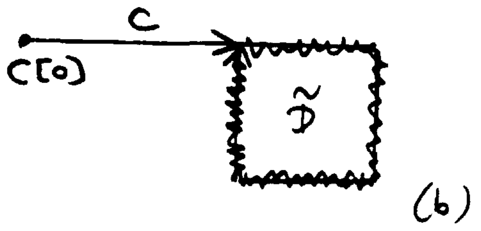


Figure 5-31: Partial Graph after path Q is finished. (a) Input to FINISH-R in line 15 of SINK-CASE. (b) Input to REACH-PS in line 17 of SINK-CASE.

Procedure SINK-CASE returns the explored graph to the calling procedure DEFICIENCY-ONE.

Theorem 13 *Given the input of a start vertex s and a deficiency-d graph G, where $d \leq 1$, the algorithm* DEFICIENCY-ONE *explores G correctly and no edge in the graph is traversed more than four times during the exploration.*

Proof: To prove the correctness of the DEFICIENCY-ONE algorithm, we show that DEFICIENCY-ONE and the procedures LOOP-CASE and SINK-CASE consider all the possible initial cases how the explorer may get stuck while taking walks. We have shown above that relocation is possible whenever needed during the algorithm. We use the correctness of the procedures FINISH and REACH-PS to argue that the DEFICIENCY-ONE algorithm returns a correctly explored graph.

DEFICIENCY-ONE is a walk-based algorithm. This means that whenever the explorer sees an unexplored edge *during a walk*, the explorer takes it. We know from Lemma 5 that the explorer loops or gets stuck at a partial source or sink during a walk-based exploration.

Lemma 4 says that a partial sink is a sink if the graph is explored by a walk-based algorithm. Therefore, any partial sink in which the explorer gets stuck during the execution of DEFICIENCY-ONE is a sink in graph G.

Taking the initial walk on the start vertex s, the explorer either gets stuck in s (line 2 of DEFICIENCY-ONE), because it loops to the partial source s, or she gets stuck in the sink v (line 15 of DEFICIENCY-ONE). In both cases, the explorer does not relocate, but starts working on a path that is headed by the vertex in which the explorer gets stuck. Any following walk may loop and end in the vertex where the explorer started from. In this case the explorer traverses the next edge on the path she is working on. If the walk does not loop back to the vertex where she started, the explorer either gets stuck at a partial source or the sink of the graph.

We know from Lemma 6 that a graph of deficiency one has at most one sink. Therefore, once the explorer gets stuck in the sink during the initial walk P, any following walk can only be a loop or end in a partial source of the graph. By Lemma 7 there is at most one partial source in G_p. Therefore, we only have to consider the cases that the explorer gets stuck in the partial source $s = P[0]$ in line 3 of procedure SINK-CASE and the case that every walk on path Q is a loop, so that $\tilde{Q}$ is finished when the explorer is stuck (line 10 of procedure SINK-CASE).

If the initial walk P from the start vertex s does not end in the sink of the graph, but in s, then there is no partial source in G_p, so every following walk is either a loop or ends in the sink of the graph. If every following walk is a loop, the graph does not have a sink. The graph has deficiency zero and is completely explored after the work on the initial walk. If the graph has deficiency one, the work on the initial walk in line 3 of DEFICIENCY-ONE ends in the sink of the graph. Again by Lemma 6 we know that once the explorer gets stuck in the sink of the graph, any following walk can only be a loop or end in the partial source of the graph. Therefore, we only consider the cases that the explorer gets stuck in the partial source $P[i]$ in line 5 of procedure LOOP-CASE and the case that every walk on path T is a loop, so that $\tilde{T}$ is finished when the explorer is stuck (line 12 of procedure LOOP-CASE).

We have shown above that the procedures FINISH, FINISH-R, and REACH-PS are called on a partial graph for which the *input assumptions* of FINISH, FINISH-R, and REACH-PS are satisfied, respectively.

We know from Section 5.2 that the procedures FINISH and FINISH-R finish exploring a deficiency-one graph given a partial graph that has a FINISH-structure. If the partial source in a partial graph is not reachable, we know from section 5.3 that the procedure REACH-PS explores the graph until the partial source is reachable and then calls procedure FINISH. It follows from Lemma 10 that the graph that is returned by procedure FINISH is explored. Thus, we conclude that the algorithm DEFICIENCY-ONE explores a deficiency-one graph correctly.

Thus, we conclude that no edge in the graph is traversed more than four times during the exploration of a deficiency-one graph by the algorithm DEFICIENCY-ONE. □

The off-line cost for traversing a deficiency-zero graph is $|E|$ (see Chapter 3). Adding an imaginary edge between sink and source into a deficiency-one graph makes an Eulerian multigraph. Therefore, there exists an Euler tour. Removing the imaginary edge from this tour gives an Eulerian path that is a path that contains every edge at least once. Thus, the off-line cost for traversing a graph of deficiency one is also $|E|$.

Theorem 13 states that the on-line cost of exploring a graph of deficiency one is at most $4|E|$. Thus, the competitive ratio for the algorithm is four.

5.5 Summary

In this chapter we presented the algorithm DEFICIENCY-ONE that solves the problem of exploring an unknown graph of deficiency one or zero. The algorithm has a competitive ratio of 4, which means that the costs of the algorithm are at most four times higher than the costs of the off-line solution.

The DEFICIENCY-ONE algorithm is a walk-based strategy. After an initial walk, the algorithm distinguishes a "loop-" and a "sink-case" depending on the outcome of the initial walk. The cases are handled by procedures LOOP-CASE and SINK-CASE which call the procedures REACH-PS, FINISH, and FINISH-R. Procedure REACH-PS is called if the initial partial source is not reachable from the sink. After the partial source is reachable, procedure REACH-PS calls procedure FINISH. Procedures FINISH and FINISH-R finish exploring the graph by calling themselves recursively.

Chapter 6

Exploring General Deficiency Graphs

In this thesis, we have carefully proven properties of graphs of deficiency-d which we have used in the deficiency-one algorithm. The deficiency-one algorithm is a combination of Deng and Papadimitriou's algorithms [DP90] and its analysis is based on Deng and Papadimitriou's ideas. The deficiency-one algorithm is interesting in its own right. However, it is important to understand the exploration problem for the deficiency-one case so that the more general exploration problem for deficiency-d graphs can be addressed.

Deng and Papadimitriou give a deficiency-d algorithm [DP90] They claim a $O(d^d)$ upper bound on the competitive ratio of their algorithm. We found their analysis proof for this algorithm to be quite terse and difficult to understand. We feel that it remains as an interesting open problem to find a *simple* algorithm and analysis for the general deficiency-d case. Since the lower bound for the exploration problem for deficiency-d graphs is $\Omega(d^2/4)$ and the gap to the $O(d^d)$ upper bound is rather large, it is an interesting open problem to find an algorithm that has a competive ratio of $O(d^m)$, for any fixed m.

6.1 Deng and Papadimitriou's Deficiency-d Algorithm

In the following, we give a brief description of a general deficiency-d algorithm. A deficiency-d graph has d sinks. In the deficiency-one algorithm, the explorer can only get stuck in a sink once, because the graph only has one sink. Whenever the explorer gets stuck afterwards, she gets stuck at a partial source. During the exploration of a general deficiency graph, however, the explorer may get stuck in partial sources and sinks in an arbitrary order.

Deng and Papadimitriou [DP90] define a path for each walk that ended in a sink; the walk that ended in the ith sink is called path P_i. The explorer tries to finish the unfinished path in the graph with the highest index i. If she gets stuck, she must move back to the path from where she took the walk. This leads to many relocations, so that the number of traversals per edge in the graph cannot shown to be polynomial in the deficiency d.

The reason why Deng and Papadimitriou choose an algorithm in which the explorer relocates to the path with the highest index is based on the following observation. The partial graph is not necessarily strongly connected, so every time the explorer creates a new path P_k, she may not be able to get back to path P_i from where she took the walk. However, she can reach edges on path P_k which is the path with the highest index in the partial graph. Therefore, she can resume exploring the graph by working on the reachable portions of path P_k.

The procedure that is used to explore the graph if the paths with the lower indices are not reachable from a newly discovered sink is essentially the REACH-PS procedure that we defined for deficiency-one graphs in section 5.3.

The analysis of a deficiency-d graph is difficult, because it involves a very careful proof of how often every edge in the graph is traversed during all the relocations that are performed.

The work on a path is interrupted when the explorer gets stuck in a new sink. When she resumes working on the path later, the path may have several finished and unfinished portions. She may then have to traverse finished portions of the path during some later work on that path. Therefore, the major task in an analysis of a deficiency d algorithm is not only to show how often every edge in the graph is traversed during all the relocations, but also how often every edge is traversed during any work on the path that contains the edge.

We have shown for the deficiency-one case that some finished parts of the partial graph can be "discarded", i.e., the explorer never traverses these parts again during the exploration. To show that paths can be discarded from the partial graph in the general deficiency case is much more difficult, because of the more complicated connectivity properties of a deficiency-d graph.

6.2 Open Questions

As mentioned above, the exploration problem for deficiency-d graphs has not been solved with an algorithm which has a competitive ratio that is polynomial in the deficiency of the graph.

Deng and Papadimitriou's introduction of the deficiency of a graph is a very useful, because it gives a parameterization for the graph exploration problem. Are there other parameterizations of the problem that lead to efficient algorithms?

In the graph model hat Deng and Papadimitriou [DP90] introduce the explorer can only "see" how many edges are going out of a vertex, but not how many edges are coming in. If we change the model so that the explorer knows the number of in-coming edges, does this extra information lead to better algorithms? Are any other changes to the exploration model useful?

As discussed in the introduction, our ultimate goal is the exploration of a real-world environment. The real world is very complicated, so we restrict ourselves to abstractions of the world. We believe that before we can approach a real-world problem, we need to be able to solve the theoretical problem. In this thesis, we presented a step towards the goal of understanding the theoretical problem. As we described above, there are still many open questions – should the graph model be changed; what are the most efficient algorithms to solve the graph exploration problem using the model Deng and Papadimitriou [DP90] proposed? The more is found out about the theoretical problem, the easier it will be to address the very difficult problem of exploring a real-world environment.

Bibliography

[Ang86] Dana Angluin. Learning regular sets from queries and counter-examples. Technical Report YALEU/DCS/TR-464, Yale University Department of Computer Science, March 1986.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[DP] Deng and Papadimitriou. private conversation.

[DP90] Xiaotie Deng and Christos H. Papadimitriou. Exploring an unknown graph. In *Proceedings of the 31st Symposium on Foundations of Computer Science*, volume I, pages 355–361, 1990.

[EJ73] Jack Edmonds and Ellis L. Johnson. Matching, Euler tours and the Chinese Postman. *Mathematical Programming*, 5:88–124, 1973.

[Hie73] Carl Hierholzer. Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. *Math. Ann.*, 6:30–32, 1873.

[KV89] Michael Kearns and Leslie G. Valiant. Cryptographic limitations on learning boolean formulae and finite automata. In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, pages 433–444, Seattle, Washington, May 1989.

[Kwa62] Meiko Kwan. Graphic programming using odd and even points. *Chinese Math.*, 1:273–277, 1962.

[RS89] Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, pages 411–420, Seattle, Washington, May 1989.