Preliminaries: Algorithms and problems

- What is an algorithm?
- Analysis of algorithms
- What is a problem?
- Algorithms and problems, computational complexity
- Efficiently solvable and intractable problems
- **NP**-completeness

What is an algorithm?

Informally, an **algorithm** is a finitely described computational procedure (e.g. a Java program or a Turing machine) that transforms an input into an output in a finite number of elementary steps and halts.[a]

An algorithm is usually specified relative to some well-defined **model of computation** (e.g. the Java programming language or Turing machines).

[a] We shall also assume that an algorithm is **deterministic**, that is, the sequence of steps is uniquely determined for each input.

Specifying algorithms

We will for the most part follow the conventions in [Jun].

- Algorithms are described using **pseudocode** (see [Jun, Section 2.4]) from which an actual implementation can be easily produced.
- For some algorithms, a high-level description is more appropriate. In these cases a more detailed pseudocode description usually either appears in [Jun] or is straightforward to produce, possibly with the help of a standard algorithms textbook such as [Cor].

Analysis of algorithms

Analyzing an algorithm is to quantify the resources (i.e. running time, memory usage, etc.) required by the algorithm.

- Analysis is conducted relative to some **model of computation** (e.g. Turing machine, Java virtual machine, ...).
- For purposes of analyzing an algorithm, we assume that the underlying model is the **random access machine (RAM)** model. See e.g. [Pap, Section 2.6].
- Resource usage is measured as a function of **input size**.
- Typically, the **worst case** behaviour is measured.

- Usually it is hard to determine the exact resource requirement of an algorithm since this depends on the low-level implementation of the algorithm and the computer architecture.
- To avoid the difficulties caused by a low-level analysis it is customary to conduct the analysis using a realistic but abstract model of computation (e.g. the RAM model) and view the elementary operations in an algorithm as simply taking constant time in the assumed model of computation.
- **Asymptotic notation** (or **rate of growth** notation) is used for indicating the resource requirement of an algorithm as a function of the input size m.
- If necessary, the analysis can later be refined to take into account the low-order terms and constants ignored by the asymptotic analysis.

Asymptotic notation

- Let $f(m)$ and $g(m)$ be nonnegative real functions defined on $\mathbb{N} = \{1, 2, 3, \ldots\}$.
- We write

$f(m) = O(g(m))$	if there exists a $c > 0$ and an $m_0 \in \mathbb{N}$ such that $f(m) \leq cg(m)$ for all $m \geq m_0$;
$f(m) = \Omega(g(m))$	if there exists a $c > 0$ and an $m_0 \in \mathbb{N}$ such that $f(m) \geq cg(m)$ for all $m \geq m_0$;
$f(m) = \Theta(g(m))$	if $f(m) = O(g(m))$ and $f(m) = \Omega(g(m))$.

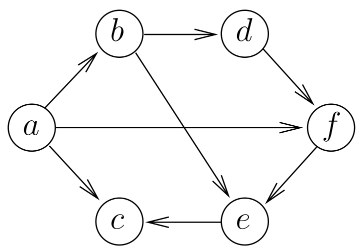
(Warning: Alternative conventions exist for asymptotic notation.)

Representing (di)graphs

- For graph algorithms, a natural notion of input size is either the number of vertices $n(G)$ or the number of edges $e(G)$ (or both) in the input (di)graph G.
- A graph G is usually given as input to an algorithm either in **adjacency list** or in **adjacency matrix** format. (Other representations for graphs include list of edges, incidence matrix, ...)
- Which representation is used depends on the application. Each representations has its advantages and disadvantages.
- Unless explicitly mentioned otherwise, we always assume that the **adjacency list** representation is used.

Adjacency list

- The vertices adjacent to each vertex are stored in a list (i.e. v occurs in the adjacency list of u if and only if $u \to v$).
- Space requirement $\Theta(e(G))$.[a]

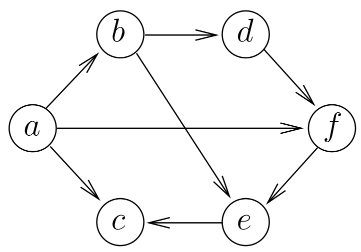


a: b, c, f
b: d, e
c:
d: f
e: c
f: e

[a] Or more accurately, $\Theta(e(G) \log n(G))$.

Adjacency matrix

- The **adjacency matrix** of a (di)graph G is the $n(G) \times n(G)$ matrix A, where $A(u, v) = 1$ if $u \to v$; otherwise $A(u, v) = 0$.
- Space requirement $\Theta(n(G)^2)$.



	a	b	c	d	e	f
a	0	1	1	0	0	1
b	0	0	0	1	1	0
c	0	0	0	0	0	0
d	0	0	0	0	0	1
e	0	0	1	0	0	0
f	0	0	0	0	1	0

What is a problem?

Informally, a **problem** (or **problem class**) consists of an infinite set of **instances** with similar structure. Associated with each instance is a set of one or more correct **solutions**. Both instances and solutions are assumed to be finite (e.g. finite binary strings).

> SHORTEST PATH: Given a graph G and two vertices $v, w \in V(G)$ as input, output a shortest path from v to w, or conclude that no such path exists.

Algorithms and problems

Algorithms can be used to solve problems.

An algorithm **solves** a problem correctly (i.e. is **correct**) if it outputs a correct solution and halts for every instance given as input.

Often the correctness of an algorithm is not immediate and a **correctness proof** is required.

Of interest is also how **efficient** an algorithm is in solving a problem. Naturally, we would like an algorithm to be as efficient as possible.

Computational complexity

Computational complexity theory studies problems with the aim of characterizing how hard (or whether at all possible) it is to solve a problem using an algorithm.

In general, the hardness of a problem depends on the underlying model of computation.

For example, it is known that deciding whether a binary string of length n is a palindrome requires $\Omega(n^2)$ time from any single-tape deterministic Turing machine.[a] On the other hand, it is easy to write a Java program that correctly detects palindromes in linear $O(n)$ time.

[a] The constant c in $\Omega(n^2)$ depends on the Turing machine used.

A central observation in computational complexity theory is that all known practically feasible universal models of computation are polynomially related: given any two such models, one can simulate the other with only a polynomial loss in efficiency. (See e.g. [Pap].)

Thus, the property whether a problem is solvable in worst case polynomial time is **independent** of the underlying model of (practically feasible) computation.

Efficiently solvable and intractable problems

An algorithm is **efficient** if its running time is bounded from above by a polynomial in the input size m. (This is naturally a very optimistic view on what is efficient. Even an algorithm with a worst case running time of, say, m^5 quickly becomes useless in practice as the input size increases.)

A problem is **efficiently solvable** (or **easy**) if there exists an efficient algorithm that solves it.

A problem for which no efficient solution algorithm *can exist* is **intractable** (or **hard**).

Both efficiently solvable and intractable problems exist.

NP-complete problems

There also exist problems for which *it is not known* whether they are efficiently solvable or intractable. The most important family of such problems is the family of **NP-complete** problems.

Many graph-theoretic problems are **NP**-complete. For example, the problem

> HAMILTONIAN CYCLE (DECISION): Given a graph G as input, decide whether G is Hamiltonian (i.e. whether G contains a spanning cycle).

is **NP**-complete.

Theorem A.1 *If any one* **NP***-complete problem is efficiently solvable, then all* **NP***-complete problems are efficiently solvable.*

No efficient algorithm for an **NP**-complete problem has been found to date, despite extensive research. Consequently, many *believe* that **NP**-complete problems are intractable.

NP-completeness and characterizing graphs

Suppose that deciding whether a graph G has property P is **NP**-complete. For example, we might choose

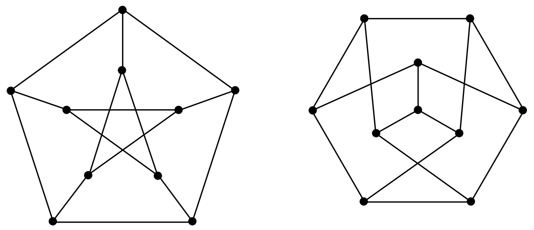
$$P = \text{“}G \text{ is Hamiltonian”} \quad \text{or} \quad P = \text{“}G \text{ is 3-colorable.”}$$

Then, **NP**-completeness theory tells us that we cannot (unless all **NP**-complete problems are efficiently solvable) give an easily testable necessary and sufficient condition for a graph to have property P.

Compare this with an easily testable property, e.g.

$$P = \text{“}G \text{ is Eulerian”} \quad \text{or} \quad P = \text{“}G \text{ is 2-colorable.”}$$

The graph isomorphism problem



There also exist problems that are *believed* to be intractable but not as hard as **NP**-complete problems. Perhaps the most important such problem is

> GRAPH ISOMORPHISM (DECISION): Given two graphs G_1 and G_2 as input, decide whether G_1 and G_2 are isomorphic.

Courses on algorithms and complexity

T-106.4100 **Design and Analysis of Algorithms**
self-explanatory

T-79.5202 **Combinatorial Algorithms**
exact and heuristic algorithms for **NP**-complete problems, computing isomorphism

T-79.1001 **Introduction to Theoretical Computer Science**
basics of Turing machines and computability theory

T-79.5103 **Computational Complexity Theory**
an advanced course on computational complexity; **NP**-completeness, approximation algorithms, randomized algorithms, intractability, . . .

Literature on algorithms

Many textbooks exist on the design and analysis of algorithms.

[Aho] A. V. Aho, J. E. Hopcroft, J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading MA, 1974.

[Cor] T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge MA, 1990.

[Sed] R. Sedgewick, P. Flajolet, *An Introduction to the Analysis of Algorithms*, Addison-Wesley, Reading MA, 1995.

Literature on computational complexity

[Sip] M. Sipser, *Introduction to the Theory of Computation*, PWS Publishing Company, Boston MA, 1997.

[Gar] M. R. Garey, D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*, W. H. Freeman and Co., San Francisco CA, 1979.

[Pap] C. H. Papadimitriou, *Computational Complexity*, Addison-Wesley, Reading MA, 1994.

[Köb] J. Köbler, U. Schöning, J. Torán, *The Graph Isomorphism Problem: Its Structural Complexity*, Birkhäuser, Boston MA, 1993.

Part II. Graph algorithms

Main reference:

[Jun] D. Jungnickel, *Graphs, Networks and Algorithms*, 2nd ed., Springer, Berlin, 2005.

Outline for part II:

1. Searching a graph, applications
2. Shortest paths and minimum spanning trees
3. Matching in bipartite and general graphs
4. Flows and circulations
5. The greedy algorithm and matroids
6. NP-complete problems and the graph isomorphism problem

1. Searching a graph, applications

Searching a graph means systematically following the edges so as to visit all the vertices [Cor].

We will consider two fundamental algorithms for searching a graph:

- breadth-first search (BFS); and
- depth-first search (DFS).

These algorithms enable us to obtain much information on the structure of a graph, which can be used to obtain an efficient (linear time) solution to many elementary graph problems.

Sources for this lecture

Elementary graph searching algorithms are discussed in almost any algorithms textbook.

The material for this lecture has been prepared with the help of [Cor, Chapter 23], [Jun, Section 3.3], and [Jun, Sections 11.2–11.5].

Breadth-first search (BFS)

Breadth-first search searches a graph G in order of increasing distance from a **source** vertex $s \in V(G)$.

Procedure BFS takes as input the pair G, s and outputs two arrays indexed by $v \in V(G)$:

$d[v]$ contains the distance $d(s, v)$; and

$p[v]$ contains a vertex that follows v in a shortest path from v to s.

We have $p[v] = \text{UNDEF}$ if either $s = v$ or no path connecting v to s exists; in the latter case also $d[v] = \infty$.

Procedure BFS(G, s; d, p)

(1) empty the queue Q;
(2) **for each** $v \in V(G)$ **do**
(3) $\quad d[v] \leftarrow \infty$;
(4) $\quad p[v] \leftarrow \text{UNDEF}$;
(5) **end for**
(6) $d[s] \leftarrow 0$;
(7) append s to Q;
(8) **while** Q is nonempty **do**
(9) $\quad v \leftarrow$ the first vertex in Q;
(10) $\quad$ remove v from Q;
(11) $\quad$ **for each** $w \in N(v)$ **do**
(12) $\quad\quad$ **if** $d[w] = \infty$ **then**
(13) $\quad\quad\quad d[w] \leftarrow d[v] + 1$;
(14) $\quad\quad\quad p[w] \leftarrow v$;
(15) $\quad\quad\quad$ append w to Q
(16) $\quad\quad$ **end if**
(17) $\quad$ **end for**
(18) **end while**

Correctness of BFS

Procedure BFS clearly halts for all inputs G, s. The following theorem shows that the content of the arrays $d[\cdot]$ and $p[\cdot]$ is as claimed when the procedure halts.

Theorem A.2 *Let $m \in \{0, 1, \ldots, \epsilon(s)\}$. Then, line (9) of Procedure BFS is executed for a vertex $v \in V(G)$ that satisfies $d(s, v) = m$. When line (9) is executed for the first time for such v, we have*

$$d[u] = \begin{cases} d(s, u) & \text{if } d(s, u) \le m\text{; and} \\ \infty & \text{otherwise.} \end{cases}$$

Moreover, the queue Q contains at that point of execution all and only vertices $u \in V(G)$ that satisfy $d(s, u) = m$.

Proof: By induction on m. The base case $m = 0$ holds when line (9) is executed for the first time. (Note that the source s is the only vertex with $d(s, u) = 0$.)

To prove the inductive step, suppose that the claim holds for m, where $m < \epsilon(s)$. Then, no vertex v with $d(s, v) = m + 1$ has been encountered so far during execution. We trace the execution further until the first vertex with $d(s, v) = m + 1$ is encountered during execution of line (9). By the inductive hypothesis, the queue Q contains all and only vertices v that satisfy $d(s, v) = m$. We show that after all these vertices are dequeued, the claim holds for $m + 1$.

Proof: (cont.) Consider any v with $d(s,v) = m$ and let $w \in N(v)$. By definition of distance, $m-1 \le d(s,w) \le m+1$. By the inductive hypothesis, we have $d[w] < \infty$ unless $d(s,w) \ge m+1$. Thus, each time the **for** loop on lines 11–17 is executed for a v with $d(s,v) = m$, only vertices $w \in N(v)$ with $d(s,w) = m+1$ are appended to the queue.

On the other hand, all $w \in V(G)$ with $d(s,w) = m+1$ will be appended to the queue since (by definition of distance) for each such w there exists a v with $w \in N(v)$ and $d(s,v) = m$. $\square$

Analysis of BFS

Each edge $vw \in E(G)$ in the component of G that contains s is considered twice on line 11 during execution of Procedure BFS.

Thus, the worst case running time of Procedure BFS is $\Theta(n(G) + e(G))$, which occurs (for example) when G is connected.

Breadth-first search trees

Theorem A.3 *Suppose Procedure BFS has been invoked on input G, s and output $p[\cdot]$ is obtained. Let T be the graph with*

$$V(T) := \{s\} \cup \{v \ :\ p[v] \ne \text{UNDEF}\},$$
$$E(T) := \{p[v]v \ :\ p[v] \ne \text{UNDEF}\}.$$

Then, T is a spanning tree for the component of G that contains s. Furthermore, $d_G(s,v) = d_T(s,v)$ for all $v \in V(T)$.

We say that T is a **breadth-first search tree** with **source** s.

Proof: We have $e(T) = n(T) - 1$ since Procedure BFS leaves $p[s] = \text{UNDEF}$. Furthermore, T is connected since there is a path from an arbitrary vertex v to the source s. Thus, T is a tree [Wes, Theorem 2.1.4].

By Theorem A.2, we have $d_G(s,v) = d_T(s,v)$ for all $v \in V(G)$ that satisfy $d_G(s,v) \le \epsilon(s)$. In particular, T spans the component of G that contains s. $\square$

Applications

Using Procedure BFS, it is straightforward to solve the following problems with worst case running time $\Theta(n(G) + e(G))$:

COMPONENTS: Output the components of G.

SHORTEST PATHS FROM SOURCE: For every $v \in V(G)$, output a shortest path from s to v if and only if v is connected to s.

SPANNING TREE: Output a spanning tree for the component of G that contains s; or, if G is connected, a spanning tree for G.

A further application: Bipartition

Procedure BFS can be modified to detect whether a graph contains an odd cycle (i.e. whether a graph is bipartite).

The following theorem tells us how odd cycles can be detected during BFS. Note that we assume G is **connected**.

Theorem A.4 *Let G be a connected graph, let $s \in V(G)$, and let T be a breadth-first search tree of G with source s. Then, G contains an odd cycle if and only if there exists an edge $uv \in E(G) \setminus E(T)$ such that $d_T(s,u) = d_T(s,v)$.*

Proof: ($\Leftarrow$) Because $d_T(s,u) = d_T(s,v)$, the path from u to v in T has even length. The edge uv completes this path into an odd cycle.

($\Rightarrow$) Let C be an odd cycle in G. Then, there exists an edge $uv \in E(C)$ such that $d_G(s,u) = d_G(s,v)$. (Exercise.) Now, since T is a breadth-first search tree with source s, we have $d_T(s,u) = d_T(s,v)$ by Theorem A.3. This implies $uv \notin E(T)$ because otherwise uv would complete a cycle in T, which is impossible. $\Box$

Thus, we can solve the following problem in worst case time $\Theta(n(G) + e(G))$.

BIPARTITION: Given a graph G as input, either output a bipartition of $V(G)$, or conclude that none exists by exhibiting an odd cycle in G.

Note that if G is not connected, then we need to apply Theorem A.4 to each component of G.

The bipartition of a component of G is given by the sets of vertices reachable by even-length and odd-length paths, respectively, from a vertex of the component. See [Jun, p. 71] for pseudocode.

Girth

BFS can be used to solve the following problem.

> GIRTH: Given a graph G, find a cycle of minimal length in G, or conclude that none exists.

The design of a solution algorithm for this problem is left as an exercise.

Hint: Consider what happens during Procedure BFS if the source s belongs to a cycle of minimal length in G. How do we detect this cycle?

Solving a maze

How do you locate the exit of a maze (assuming that you are in the maze, on your own, and equipped with, say, a magic marker or a large supply of pebbles for keeping track of progress)?

Preferably, you would like to walk as little as possible, so breadth-first search is not a good solution (why?).

The obvious strategy is to explore the maze until either

1. a dead end; or
2. an already explored part of the maze is encountered.

When this happens, turn around and backtrack to the most recently visited intersection with an unexplored choice and continue.

This strategy is **depth-first search** in action:
The maze can be viewed as a graph where each intersection is a vertex and the edges represent passages between intersections.
For an example, see [Jun, Exercise 11.2.6 on p. 337 and p. 525–527].

Depth-first search can be considered in a sense "optimal" strategy in terms of the length of the required walk (why?).

Depth-first search (DFS) on a digraph

Let G be a **directed** graph. (We will return to DFS on undirected graphs later.)

Procedure DFS takes as input a digraph G and outputs three arrays indexed by $v \in V(G)$:

$d[v]$	contains the discovery time of v;
$f[v]$	contains the finishing time of v;
$p[v]$	contains either the vertex that precedes v in the search or UNDEF if no predecessor exists.

A vertex is **discovered** when DFS first encounters it. A vertex is **finished** when all of its outgoing edges are explored.

Below is a recursive implementation of depth first search.

Procedure DFS(G; d,f,p)

(1) **for each** $v \in V(G)$ **do**
(2) $\quad d[v] \leftarrow$ UNDEF;
(3) $\quad f[v] \leftarrow$ UNDEF;
(4) $\quad p[v] \leftarrow$ UNDEF
(5) **end for**
(6) $t \leftarrow 1$;
(7) **for each** $v \in V(G)$ **do**
(8) $\quad$ **if** $d[v] =$ UNDEF **then**
(9) $\qquad$ DFS-VISIT(v)
(10) $\quad$ **end if**
(11) **end for**

Procedure DFS-VISIT(v)

(1) $d[v] \leftarrow t$;
(2) $t \leftarrow t+1$;
(3) **for each** $w \in N^+(v)$ **do**
(4) $\quad$ **if** $d[w] =$ UNDEF **then**
(5) $\qquad p[w] \leftarrow v$;
(6) $\qquad$ DFS-VISIT(w)
(7) $\quad$ **end if**
(8) **end for**
(9) $f[v] \leftarrow t$;
(10) $t \leftarrow t+1$

The counter variable t and the arrays $d[\cdot], f[\cdot], p[\cdot]$ are assumed to be accessible from Procedure DFS-VISIT, which performs the actual search.

Correctness and analysis of DFS

Procedure DFS halts for every input since DFS-VISIT is called exactly once for each vertex $v \in V(G)$. In particular, each edge $vw \in E(G)$ is explored exactly once on line 3 of DFS-VISIT.

The worst case running time of Procedure DFS is therefore $\Theta(n(G) + e(G))$.

Arborescence, root

DFS has a very useful concept analogous to breadth-first search trees in BFS. We require some preliminary definitions.

Let G be a digraph. A vertex r is a **root** in G if there exists a path from r to v for each $v \in V(G)$.

An oriented tree that has a root is called an **arborescence**.

Predecessor subgraph, ancestor, descendant

Let G be a digraph. Suppose Procedure DFS is invoked with input G and output $d[\cdot], f[\cdot], p[\cdot]$ is obtained. We say that the subgraph P defined by

$$V(P) := V(G),$$
$$E(P) := \{p[v]v \; : \; p[v] \neq \text{UNDEF}\}$$

is a **predecessor subgraph** of G.

Let $u, v \in V(G)$. We say that v is an **descendant** of u (or, equivalently, u is a **ancestor** of v) if there exists a path from u to v in P. An ancestor/descendant is **proper** if $u \neq v$.

It is straightforward to show (cf. Theorem A.3) that the predecessor subgraph P is a vertex-disjoint union of arborescences. The roots of the maximal arborescences in P are precisely the vertices v with $p[v] = \text{UNDEF}$.

The graph P is sometimes called a **depth-first search forest**; similarly, the maximal arborescences in P are **depth-first search trees**. Note that this is somewhat inaccurate because trees and forests are by definition undirected graphs.

Classification of edges

It will be useful to classify the edges in G into types based on the predecessor subgraph P:

tree edges	are edges in P;
back edges	are edges uv that connect u to an ancestor v;
forward edges	are nontree edges uv that connect u to a proper descendant v;
cross edges	are all other edges in G.

The edge classification can be performed as the edges are explored during DFS with the help of the arrays $d[\cdot]$ and $f[\cdot]$ (exercise).

Properties of DFS

We establish some properties of DFS before discussing its applications.

The following two observations are immediate corollaries of the structure of DFS-VISIT.

Theorem A.5 *Let $u, v \in V(G)$ such that $d[u] < d[v]$. Then, either*

$$d[u] < f[u] < d[v] < f[v] \qquad \text{or} \qquad d[u] < d[v] < f[v] < f[u].$$

Theorem A.6 *Let $u, v \in V(G)$. Then, v is a descendant of u if and only if $d[u] \le d[v] < f[v] \le f[u]$.*

A vertex $w \in V(G)$ is **undiscovered** if $d[w] = \text{UNDEF}$.

Theorem A.7 *Let $u, v \in V(G)$. Then, v is a descendant of u if and only if at the time* DFS-VISIT *is invoked with input u, there exists a path from u to v in G consisting of undiscovered vertices only.*

Proof: ($\Rightarrow$) By Theorem A.6 we have $d[u] \le d[w] < f[w] \le f[u]$ if and only if w is a descendant of u. Thus, the path from u to v in P consists of vertices with $d[w] = \text{UNDEF}$ at the time DFS-VISIT is invoked with input u.

Proof: ($\Leftarrow$) Let $u = w_1, w_2, \ldots, w_n = v$ be the vertices of a path from u to v consisting of undiscovered vertices only. Clearly, $w_1 = u$ is a descendant of u. If $n = 1$, we are done. Otherwise, suppose w_j is a descendant of u, where $1 \le j < n$. We prove that w_{j+1} is a descendant of u. Because w_j is a descendant of u, we have $d[u] \le d[w_j] < f[w_j] \le f[u]$. Since w_{j+1} is undiscovered when DFS-Visit is invoked with u, we must have $d[u] < d[w_{j+1}]$. There are two cases to consider. If $d[w_{j+1}] < d[w_j]$, then $d[u] < d[w_{j+1}] < f[u]$, so $d[u] < d[w_{j+1}] < f[w_{j+1}] < f[u]$ by Theorem A.5. On the other hand, if $d[w_j] < d[w_{j+1}]$, then $w_{j+1} \in N^+(w_j)$ implies that the call DFS-Visit(w_{j+1}) must finish before the call DFS-Visit(w_j). Hence, $f[w_{j+1}] \le f[w_j]$. Combining inequalities, $d[u] \le d[w_{j+1}] < f[w_{j+1}] \le f[w_j] \le f[u]$. Therefore, in both cases w_{j+1} is a descendant of u by Theorem A.6. $\square$

Applications of DFS on digraphs

We will consider three standard applications of DFS on digraphs:

- determining whether a digraph is acyclic;
- topologically sorting the vertices of an acyclic digraph; and
- determining the strong components of a digraph.

Each of these problems has a linear time (i.e. $O(n(G) + e(G))$) solution using DFS.

Detecting cycles with DFS

A digraph is **acyclic** if it does not contain a cycle.

Note: loops are cycles.

Theorem A.8 *A digraph is acyclic if and only if there are no back edges.*

Proof: Exercise. $\square$

Topological sort

Let G be a digraph. A **topological sort** of G is a linear order "$\prec$" on $V(G)$ that satisfies $u \prec v$ for every edge $uv \in E(G)$.

Theorem A.9 *A topological sort of G exists if and only if G is acyclic.*

Proof: ($\Rightarrow$) If G contains a cycle, then clearly no linear order on $V(G)$ is a topological sort.

($\Leftarrow$) We can obtain a topological sort for any acyclic digraph using DFS; this is the content of the following theorem. $\square$

Theorem A.10 *Let G be an acyclic digraph. Then, $f[v] < f[u]$ for any edge $uv \in E(G)$.*

Proof: Let $uv \in E(G)$. Loops cannot occur in an acyclic graph, so $u \neq v$. If $d[u] < d[v]$, then v becomes a descendant of u by Theorem A.7 since both u and v are undiscovered when DFS-VISIT is invoked with input u. Hence, $f[v] < f[u]$.

If $d[v] < d[u]$, we cannot have $f[u] < f[v]$ because then u would be a descendant of v and uv would be a back edge, which is impossible by Theorem A.9. Hence, $f[v] < f[u]$ also when $d[v] < d[u]$. $\square$

Thus, the linear order "$\prec$" on $V(G)$ defined by $u \prec v$ if and only if $f[v] < f[u]$ is a topological sort of G.

Strong components

Recall that a **strong component** in a digraph G is a maximal strongly connected subgraph. Moreover, each vertex in G belongs to a unique strong component, and two vertices $u, v \in V(G)$ are in the same strong component if and only if there exist paths from u to v and from v to u.

The following algorithm for computing strong components using DFS appears in [Cor, Section 23.5] and [Jun, Section 11.5].
Aho, Hopcroft, and Ullman (1983) attribute this algorithm to R. Kosaraju (1978, unpublished) and M. Sharir (1981).

An alternative algorithm is due to Tarjan (1972).

Before describing the algorithm, we characterize the strong components in a digraph using DFS.

Suppose DFS is run on the digraph G. Associate with each vertex $u \in V(G)$ the vertex $\phi(u)$ (the **forefather** of u) that has the largest finishing time among the vertices reachable from u in G.

In other words, $\phi(u)$ is the unique vertex for which there exists a path from u to $\phi(u)$ and the inequality $f[w] \leq f[\phi(u)]$ holds for all vertices w reachable from u.

Theorem A.11 *Let $u \in V(G)$. Then, u is a descendant of $\phi(u)$.*

Proof: There exists a path from u to $\phi(u)$ by definition of a forefather. Denote by t the first vertex discovered from this path during DFS. Then, $\phi(u)$ becomes a descendant of t by Theorem A.7. Consequently, $f[\phi(u)] \leq f[t]$. Since there is a path from u to t, we must have $f[\phi(u)] \geq f[t]$ by definition of a forefather. So, $\phi(u) = t$ and $\phi(u)$ is the first vertex discovered from the path. In particular, $d[\phi(u)] \leq d[u]$. Thus, u is a descendant of $\phi(u)$ since $f[u] \leq f[\phi(u)]$ by definition of a forefather. $\square$

Corollary A.1 *Let $u \in V(G)$. Then, u is in the same strong component with its forefather $\phi(u)$.*

Proof: Clear by definition of a forefather and Theorem A.11. □

Corollary A.2 *Let $u, v \in V(G)$. Then, u and v are in the same strong component if and only if $\phi(u) = \phi(v)$.*

Proof: ($\Rightarrow$) Since u and v are in the same strong component, a vertex w is reachable from u if and only if it is reachable from v. Hence, by definition of a forefather, $\phi(u) = \phi(v)$.

($\Leftarrow$) By definition of a forefather, there exists a path from u to $\phi(u)$. Since v is a descendant of $\phi(v)$ (Theorem A.11), there exists a path from $\phi(v)$ to v. Thus, $\phi(u) = \phi(v)$ implies that there exists a path from u to v. We obtain a path from v to u by exchanging the roles of u and v in the above argument. □

The observations below form the basis of the algorithm.

- The vertex v that finishes last in DFS must be a forefather since its finishing time is the maximum in $V(G)$.
- By Corollaries A.1 and A.2, the strong component of v consists of precisely the vertices that can reach v.
- Equivalently, the strong component of v consists of precisely the vertices *reachable from* v when the direction of each edge has been reversed.

The following procedure computes the strong components of G.

Procedure STRONG(G)

run DFS on G;
reverse the direction of all edges in G;
while G is nonempty **do**
    determine the vertex $v \in V(G)$ for which $f[v] = \max_{w \in V(G)} f[w]$;
    compute the vertices $S \subseteq V(G)$ reachable from v;
    report S as a strong component;
    delete the vertices in S from G
end while

This procedure can be implemented so that it runs on worst case time $\Theta(n(G) + e(G))$.

In practice, the algorithm of Tarjan (1972) is more efficient for computing strong components.

Depth-first search on undirected graphs

Depth-first search on undirected graphs is in many ways simpler than on digraphs.

Procedure DFS works on an undirected graph G if we replace "$w \in N^+(v)$" with "$w \in N(v)$" on line (3) of DFS-VISIT.

It will be useful to insist that the predecessor graph P obtained from DFS is a digraph even though G is undirected. In particular, the concepts of a descendant and ancestor remain well-defined in this case.

With this assumption Theorems A.6 and A.7 are valid also in the undirected case.

Edge classification in the undirected case

Let G be a graph and let P be the predecessor digraph obtained from Procedure DFS.

Theorem A.12 *For every undirected edge $uv \in E(G)$, either u is a descendant of v or v is a descendant of u.*

Proof: The claim is an immediate consequence of Theorem A.7. □
Thus, cross edges do not exist in the undirected case.

An edge of G is a **tree edge** if it belongs to the graph underlying P. All other edges in G are **back edges** (since DFS explores a nontree edge first in the direction from descendant to ancestor).

Applications of DFS on undirected graphs

- A graph G is acyclic if and only if DFS on G produces no back edges. (Theorem A.8 holds also in the undirected case.)
- A graph G is connected if and only if the predecessor graph P has exactly one root. (This is an immediate consequence of Theorem A.7.)
- The cut-vertices, cut-edges, and blocks of a graph can be computed using DFS.

Cut-vertex, cut-edge, block

A vertex (edge) of a graph G is a **cut-vertex** (**cut-edge**) if its removal increases the number of components in G.

A **block** of G is a maximal connected subgraph of G that has no cut-vertex.

Example. See [Wes, Example 4.1.17].

A graph G is **2-connected** if it is connected, has no cut-vertex, and has at least three vertices.

Properties of blocks

Two distinct blocks of G may have at most one vertex in common [Wes, Proposition 4.1.9].

A vertex $v \in V(G)$ is a cut-vertex of G if and only if there exist two blocks B_1, B_2 such that $V(B_1) \cap V(B_2) = \{v\}$.

A block of G with two vertices is a cut-edge of G. Every edge in G occurs in a unique block.

Identifying cut-vertices and cut-edges using DFS

Let G be a graph and suppose DFS is invoked with input G.

Let $\ell(u)$ be the set of vertices consisting of the vertex u and all vertices v such that there exists a back edge wv, where w is a descendant of u. Denote by $L(u)$ the minimum value of $d[v]$ among the vertices $v \in \ell(u)$.

Clearly,

$L(u) = \min \{d[u]\} \cup \{L(w) \ : \ p[w] = u\} \cup \{d[v] \ : \ uv \text{ is a back edge}\}.$

Thus, $L(u)$ can be computed during DFS (see [Jun, p. 342–343]).

Theorem A.13 *A nonroot vertex u is a cut-vertex if and only if there exists a tree edge uv (where $d[u] < d[v]$) such that $L(v) \geq d[u]$.*

Proof: ($\Rightarrow$) We may assume that G is connected. (Otherwise consider the component that contains u.) Let s be the root vertex and let $V_1, \ldots, V_k$ be the vertex sets of the components that result if u is removed. Suppose $s \in V_1$ and let uv be the first edge explored by DFS such that $v \notin V_1$ (say, $v \in V_2$) and $u \neq v$. Clearly, uv becomes a tree edge and $d[u] < d[v]$. Furthermore, all and only vertices in V_2 become descendants of v since u is a cut-vertex. Hence, $L(v) \geq d[u]$ because an edge with only one endvertex in V_2 must have u as the other endvertex.

Proof: ($\Leftarrow$) Denote by D the set of descendants of v. Let xy be an edge with exactly one endvertex in D, say $x \in D$. Then, v is a proper descendant of y by Theorem A.12. Because $L(v) \geq d[u]$ and uv is a tree edge, we must have $y = u$. Thus, all paths from D to the complement of D contain u. Since $s \notin D$ and $s \neq u$, removing u disconnects s from v. Hence, u is a cut-vertex. □

Theorem A.14 *A root s is a cut-vertex if and only if s is incident with more than one tree edge.*

Proof: Exercise. □

A linear time algorithm for computing the cut-vertices and blocks of a graph G that relies on these observations appears in [Jun, Algorithm 11.3.8]. The algorithm is due to Tarjan (1972).