10.1 SEARCHING

628min 7–8–2003 *Harold N. Gabow, University of Colorado*

INTRODUCTION

A ***search*** of a graph is a methodical exploration of all the vertices and edges. It must run in "linear time", i.e., in one pass (or a small number of passes) over the graph. Even with this restriction, a surprisingly large number of fundamental graph properties can be tested and identified.

This section examines the two most important search methods. *Breadth-first search* gives an efficient way to compute distances. *Depth-first search* is useful for checking many basic connectivity properties, for checking planarity, and also for data flow analysis for compilers. A treatment of at least some aspects of both these methods can be found in almost any algorithms text (some recent ones are [BrBr96, CLRS01, GoTa02, HSR98, Se02, We99]).

All the algorithms of this section (except for §10.1.7) run in linear time or very close to it. Since it takes linear time just to read the graph, the algorithms are essentially as efficient as possible (they are "asymptotically optimal").

NOTATION: Throughout this chapter, the number of vertices and edges of a graph $G = (V, E)$ are denoted n and m, respectively. Time bounds for algorithms are given using asymptotic notation, e.g., $O(n)$ denotes a quantity that, for sufficiently large values of n, is at most cn, for some constant c that is independent of n.

CONVENTION: In all algorithms, we assume that the graph G is given as an adjacency list representation. If G is undirected, this means that each vertex has a list of all its neighbors. The list can be sequentially allocated or linked. If G is directed, then each vertex has a list of all its out-neighbors.

10.1.1 Breadth-First Search

The ***breadth-first search*** method (abbr. ***bfs***) finds shortest paths from a given vertex of a graph to other vertices. It generalizes to Dijkstra's algorithm, which allows numerical (nonnegative) edge-lengths. Throughout this section, the given graph G can be directed or undirected.

DEFINITIONS

D1: A ***length function*** on a graph specifies the numerical length of each edge. Each edge is assumed to have length one, unless there is an explicitly declared length function.

D2: The ***distance*** from vertex u to vertex v in a graph, denoted $d(u, v)$, is the length of a shortest path from u to v.

D3: The ***diameter*** of a graph is the maximum value of $d(u, v)$ for $u \neq v$.

D4: A ***shortest-path tree*** T from a vertex s is a tree, rooted at s, that contains all the vertices that are reachable from s. The path in T from s to any vertex x is a shortest path in G, i.e., it has length $d(s, x)$.

EXAMPLES

E1: Figure 10.1.1 gives a shortest path tree from vertex s.

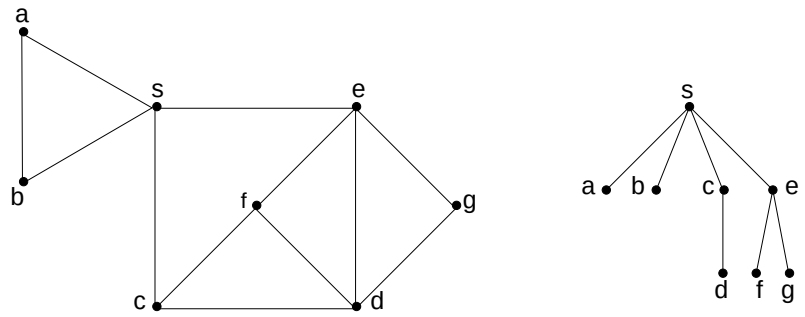

Figure 10.1.1 Undirected graph and shortest path tree.

E2: The *small-world phenomenon* [Mi67, Kl00] occurs when relatively sparse graphs have low diameter. Studies have shown that the graphs of movie actors, neural connections in the *c.∼elegans* worm, and the electric power grid of the western United States all exhibit the small-world phenomenon. The world-wide web is believed to have this structure too.

E3: For several decades, mathematicians have computed their *Erdös number* as their distance from the prolific mathematician Paul Erdös, in the graph where an edge joins two mathematicians who have coauthored a paper.

E4: The premise of the *6 Degrees of Kevin Bacon* game is that the graph whose vertices are movie actors and whose edges join two actors appearing in the same movie has diameter at most 6.

E5: In computer and communications networks, a message is typically broadcast from one site s to all others by passing it down a shortest path tree from s.

E6: To solve a puzzle like Sam Lloyd's "15 puzzle", we can represent each position by a vertex. A directed edge (i, j) exists if we can legally move from i to j. We seek a shortest path from the initial position to a winning position.

Ordered Trees

DEFINITIONS

D5: In a rooted tree, a vertex x is an ***ancestor*** of a vertex y, and y is a ***descendant*** of x, if there is a path from x to y whose edges all go from parent to child. By convention x is an ancestor and descendant of itself (e.g., in the tree of Figure 10.1.1 vertex e has 3 descendants).

D6: Vertex x is a ***proper ancestor (descendant)*** of vertex y if it is an ancestor (descendant) and $x \neq y$.

D7: An ***ordered tree*** is a rooted tree in which the children of each vertex are linearly ordered. In a plane drawing of such a tree, left-to-right order gives the order of the children. (The leftmost child is first.)

D8: Vertex x is ***to the left of*** vertex y if some vertex has children c and d, with c to the left of d, c an ancestor of x and d an ancestor of y.

D9: In a graph G, a ***breadth-first tree*** T from a vertex s contains the vertices that are reachable from s. It is an ordered tree, rooted at s. If x is a vertex at depth δ in the tree T, then the children of x in T are the vertices of G that are adjacent in G to x, but not adjacent (in G) to any vertex in T at depth less than δ, or to any vertex at depth δ in T that is at the left of x.

FACTS

F1: Any breadth-first tree is a shortest-path tree.

F2: A high level bfs algorithm is given below as Algorithm 10.1.1. It constructs a breadth-first tree. It starts from s, finds the vertices at distance 1 from s, then the vertices at distance 2, etc.

Algorithm 10.1.1: Breadth-first Search

Input: directed or undirected graph $G = (V, E)$, vertex s
Output: breadth-first tree T from s
$V_i = \{$all vertices at distance i from $s\}$

```
V_0 = {s}
make s the root of T
i = 0
while V_i ≠ ∅ do /* construct V_{i+1} */
    V_{i+1} = ∅
    for each vertex v ∈ V_i do
    /* "scan" v */
        for each edge (v, w) do
            if w ∉ ⋃_j V_j then
                make w the next child of v in T
                add w to V_{i+1}
    i = i + 1
```

F3: The high-level algorithm can be implemented to run in total time $O(n+m)$. The main data structure is a queue of vertices that have been added to T, but whose children in T have not been computed.

F4: In general we verify that an algorithm takes time $O(n+m)$ by checking that it spends constant time (i.e., $O(1)$ time) on each vertex and edge of G.

F5: Not every shortest path tree is a breadth-first tree (e.g., the tree of Figure 10.1.1). This does not cause any problems in applications.

F6: The diameter can be found by doing a breadth-first search from each vertex.

F7: Dijkstra's algorithm computes a shortest path tree from s in a graph with a nonnegative length function. It generalizes breadth-first search. Like bfs it finds the set V_d of all vertices at distance d from s, for increasing values of d. An appropriate data structure implements the algorithm in time $O(m + n \log n)$ [FrTa87, CLRS01].

10.1.2 Depth-First Search

Depth-first search (abbr. ***dfs***) was investigated in the 19^{th} century as a strategy for exploring a maze [Lu82, Tarr95]. The fundamental properties of the depth-first search tree were discovered by Hopcroft and Tarjan [HoTa73a, Ta72]. Tarjan also developed many other elegant and efficient dfs algorithms (see §10.1.6). The idea of depth-first search is to scan repeatedly an edge incident to the most recently discovered vertex that still has unscanned edges.

DEFINITIONS

D10: Two vertices in a tree are ***related*** if one is an ancestor of the other.

D11: In an undirected graph $G = (V, E)$, a ***depth-first tree*** (abbr. ***dfs tree***) from a vertex s is a tree subgraph T, rooted at s, that contains all the vertices of G that are reachable from s.

- Edges of $E(T)$ and $E(G) - E(T)$ are called ***tree edges*** and ***nontree edges***, respectively.
- Each nontree edge is also called a ***back edge***.

The crucial property is that the two endpoints of each back edge are related.

D12: In an undirected graph G, a ***depth-first spanning forest*** is a collection of depth-first trees, one for each connected component of G. Each vertex of G belongs to exactly one tree of the forest.

D13: Let $G = (V, E)$ be a directed graph where every vertex is reachable from a designated vertex s. A ***depth-first tree*** from s is an ordered tree in G, rooted at s that contains all vertices V. Each edge of T is called a ***tree edge***. Each ***nontree edge*** $(x, y) \in E - T$ can be classified into one of three types:

- A ***back edge*** has y an ancestor of x.
- A ***forward edge*** has y a descendant of x.
- A ***cross edge*** joins two unrelated vertices.

The crucial property is that each cross edge (x, y) has x to the right of y.

D14: Let G be a directed graph, in which we no longer assume that some vertex can reach all others. A ***depth-first forest*** is an ordered collection of trees in G so that each vertex of G belongs to exactly one tree. The edges of G are classified into the 4 types of edges in Definition 13 with one additional possibility:

- A ***cross edge*** can join 2 vertices in different trees as long as it goes from right to left (i.e., from a higher numbered tree to a lower numbered tree).

EXAMPLES

E7: Figure 10.1.2 illustrates a depth-first search of an undirected graph. In drawings of depth-first spanning trees, tree edges are solid and nontree edges are dashed. There can be many depth-first trees with the same root. For instance the tree edge $(5, 6)$ could be replaced by $(5, 7)$.

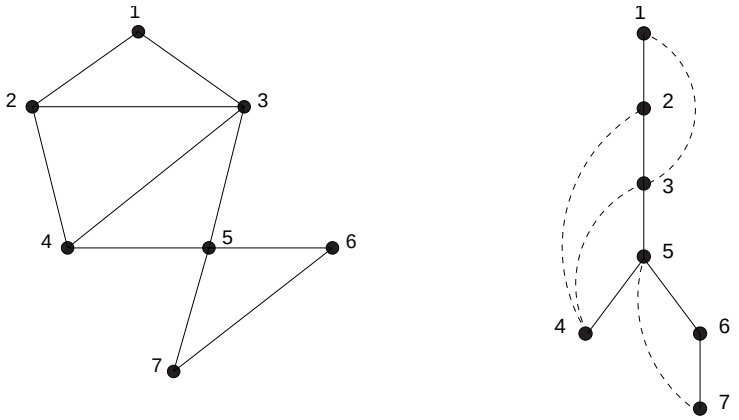

Figure 10.1.2 Undirected graph and depth-first spanning tree.

E8: Figure 10.1.3 illustrates a depth-first search of a directed graph. There is 1 forward edge, 2 back edges and 2 cross edges.

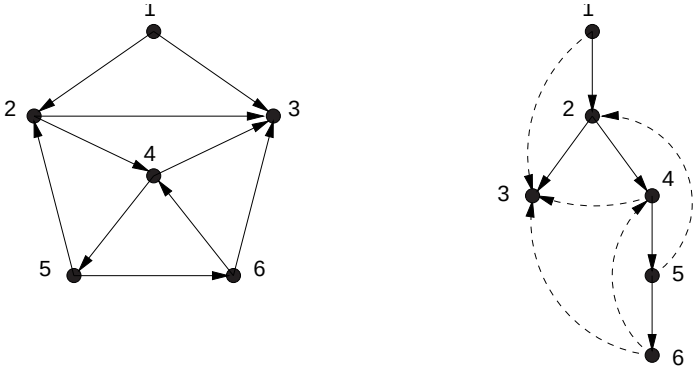

Figure 10.1.3 Directed graph and depth-first spanning tree.

FACTS

F8: Any vertex s of an undirected graph has a depth-first tree from s. Any vertex s of a directed graph has a depth-first tree of the subgraph induced by the vertices reachable from s. A high level algorithm to find such a tree is the following.

Algorithm 10.1.2: Depth-First Search

Input: directed or undirected graph $G = (V, E)$, vertex s
Output: depth-first tree T from s

make s the root of T
`DFS`(s)

procedure `DFS`(v)
/* vertex v is ***discovered*** at this point */
for each edge (v, w) **do**
    /* edge (v, w) is ***scanned (from*** v***)*** at this point */
    if w has not been discovered **then**
        make w the next child of v
        `DFS`(w)
/* vertex v is ***finished*** at this point */

F9: The procedure `DFS` is recursive, i.e., it calls itself. The overhead for a recursive call is $O(1)$. Algorithm 10.1.2 uses linear time, $O(n + m)$.

F10: If scanning edge (v, w) from the vertex v results in the discovery of the vertex w, then (v, w) is a tree edge.

F11: Suppose that the graph G is undirected. For the tree T produced by Algorithm 10.1.2 to be a valid depth-first tree, any edge $(v, w) \in E - T$ must have v and w related vertices. Why does T have this property? By symmetry suppose v gets discovered before w. Then w will either be made a child of v (like edge $(3, 5)$ in Figure 10.1.2) or a nonchild descendant of v (like edge $(3, 4)$ in Figure 10.1.2).

F12: Suppose that the graph G is directed. For T to be a valid depth-first tree, any edge (v, w) must be one of the 4 possible types. Why does T have this property? First suppose v gets discovered before w. In that case w will be a descendant of v and (v, w) will be a tree or forward edge (as in Fact 11). Next suppose v is discovered after w. Then either v descends from w or v is to the right of w. In the former case (v, w) is a back edge and in the latter case (v, w) is a cross edge.

F13: Algorithm 10.1.2 can be extended to a procedure that constructs a depth-first forest F: The procedure starts with $F = \emptyset$. It repeatedly chooses a vertex $s \notin F$, uses `DFS`(s) to grow a depth-first tree T from s, and adds T to F.

F14: Algorithm 10.1.2 uses linear time. (For directed graphs a point to note is that a vertex w gets added to only 1 tree of F. This is because once discovered, vertex w remains "discovered" throughout the whole procedure.)

REMARKS

R1: We can test whether an undirected graph is connected in linear time, by using a depth-first search. The trees of a depth-first search spanning forest give the connected components.

R2: We can test whether all vertices of a directed graph are reachable from a vertex s in linear time, by a depth-first search.

Discovery Order

DEFINITIONS

D15: ***Discovery order*** is a numbering of the vertices from 1 to n in the order they are discovered. This is also called the ***preorder*** of the dfs tree.

D16: In ***finish time order*** the vertices are numbered from 1 to n by increasing finish time. This is the ***postorder*** of the dfs tree.

FACTS

F15: Most algorithms based on the depth-first search tree use discovery order. These algorithms identify each vertex v with its discovery number, also called v. This is how the vertices are named in Figure 10.1.3.

F16: In discovery order, the descendants of a vertex v are numbered consecutively, with v first, followed by all its proper descendants. This gives a quick way to test if a given vertex w descends from another given vertex v: Let v have d descendants. w is a descendant of v exactly when $v \leq w < v + d$. This method can be implemented to run in $O(1)$ (i.e., constant) time.

REMARKS

R3: The power of depth-first search comes from its simplification of the edge structure – the absence of cross edges in undirected graphs, and the absence of left-to-right edges in directed graphs. Depth-first search algorithms work by propagating information up or down the dfs tree(s).

R4: Many simple properties of graphs can be analyzed without using the full power of depth-first search. The algorithm always works with a path in the dfs tree, rather than with the entire dfs tree. The algorithm propagates information along the path.

R5: As a simple example of Remark 4 we give a procedure that shows an undirected graph with minimum degree δ has a path of length $> \delta$: execute `DFS`(s) (for any s), stopping at the first vertex t that becomes finished. The portion of tree T constructed by this procedure is a path from s to t of length $> \delta$. The reason is that all of t's neighbors must be in the path for t to be finished.

R6: Sections 10.1.3–5 deal with simpler graph properties that can be handled by the path view of depth-first search. §10.1.6 covers deeper properties whose algorithms require the full power of the depth-first search tree. §10.1.7 deals with both views of depth-first search.

10.1.3 Topological Order

Topological order is the fundamental property of directed acyclic graphs. In conjunction with dynamic programming, topological order leads to efficient algorithms for many fundamental properites of directed acyclic graphs – even properties that are NP-complete in general graphs.

DEFINITIONS

D17: A ***dag*** is a directed acyclic graph, i.e., it has no cycles.

D18: A ***source*** (***sink***) of a dag is a vertex with indegree (outdegree) 0.

D19: A ***topological numbering (topological order, topological sort)*** of a directed graph assigns an integer to each vertex so that each edge is directed from lower number to higher number.

EXAMPLES

E9: The dag of Figure 10.1.4 has source a and sink f. Alphabetic order is a valid topological ordering. In general a dag has many topological numberings. In this figure 12 are possible.

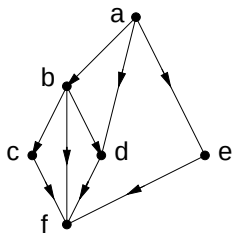

Figure 10.1.4 Dag and topological order.

E10: A dag can always be drawn so that all edges are directed downwards, as in Figure 10.1.4. Topological numbers guide the vertical placement of the vertices. This principle is useful in algorithms for drawing graphs (see Section 10.3).

E11: Prerequisite graphs in a university department are dags: if course X is a prerequisite to course Y, then an arrow is drawn from X to Y. There cannot be a cycle, else no one could graduate! The course numbering is a topological numbering: a prerequisite to a course always has a lower number.

E12: A combinational circuit is a collection of logic gates and interconnecting wires, with no feedback. The no-feedback property makes it a dag.

E13: A graph of program dependencies is a dag (assuming no recursion is allowed). For instance the dependencies specified by a `makefile` is a dag. The `make` utility always ensures that a file's timestamp is no later than the timestamp of any dependent file. Thus the timestamps form a topological numbering.

E14: The formulas in a spreadsheet depend on one another, and this dependence relation is a dag. When the value of a cell is changed, the values of dependent cells are recalculated in topological order.

E15: In ecology, a *food web* is a graph whose vertices are the species of an ecosystem. An arrow is drawn from one species to all the other species it preys upon. This model is commonly assumed to be a dag, to disallow cycles in the food chain.

FACTS

F17: Every dag has one or more sources and one or more sinks. This can be seen by examining a path of maximal length. The first (last) vertex must be a source (sink), since otherwise the path could be extended at the beginning (end).

F18: A graph with a topological numbering is a dag. To see this observe that topological numbers increase along a path, so a path cannot return to its starting vertex. Thus no cycle exists.

F19: Any dag has a topological numbering. To construct such a numbering with lowest number 1, assign the lowest number to a source s. Then proceed recursively on the dag $G - s$, using lowest number 2.

F20: One can similarly construct a topological numbering by repeatedly numbering a sink s with the highest number, and proceeding recusively on dag $G - s$.

F21: The strategy of Fact 20 can be implemented efficiently by depth-first search. The reason is that as we grow a depth-first path in a dag, the first vertex to become finished is a sink. More succinctly, we can grow a depth-first path until a sink is reached. This gives the following high-level algorithm.

Algorithm 10.1.3: Topological Numbering (High Level)

Input: dag $G = (V, E)$
Output: topological numbering of G: vertex v has number $I[v]$

repeat until G has no vertices:
    grow a dfs path P until a sink s is reached
    set $I[s] = n$, decrease n by 1 and delete s from P & G

To make this algorithm efficient, each iteration grows the dfs-path P by starting with the previous P and extending it, if possible.

F22: A lower level implementation of Algorithm 10.1.3 runs in linear time. The idea is to use array $I[1..n]$ for 2 purposes:

$$I[v] = \begin{cases} 0 & \text{if } v \text{ has never been in } P \\ t & \text{if } v \text{ has been deleted and assigned topological number } t \end{cases}$$

Algorithm 10.1.4: Topological Numbering (Lower Level)

Input: dag $G = (V, E)$
Output: topological numbering of G: vertex v has number $I[v]$

$num = n$;
for each vertex v **do** $I[v] = 0$
for each vertex v **do if** $I[v] = 0$ **then** `DFS`(v)

procedure `DFS`(v)
for each edge (v, w) **do**
    if $I[w] = 0$ **then** `DFS`(w)
/* v is now a sink in the high level algorithm */
$I[v] = num$; decrease num by 1
/* v is now deleted in the high level algorithm */

F23: Algorithm 10.1.4 runs in linear time. It spends $O(1)$ time on each vertex and edge.

EXAMPLE

E16: Figure 10.1.5 illustrates how the algorithm numbers the dag of Figure 10.1.4.

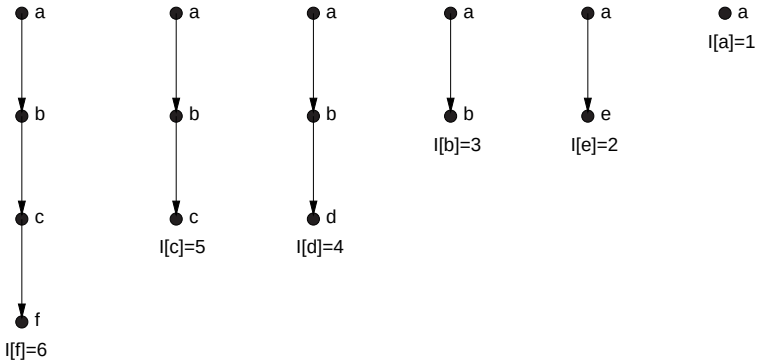

Figure 10.1.5 Execution of topological numbering algorithm.

FACTS

F24: Listing the vertices in order of decreasing finish time (Definition 16) is a valid topological order.

F25: Tarjan's algorithm for topological order [Ta74b, CLRS01] is based on Fact 24. Algorithm 10.1.4 is a reinterpretation of Tarjan's algorithm.

To illustrate this, Figure 10.1.6 shows a dfs tree for Figure 10.1.4. Each vertex is labelled by its name and finish number. Subtracting each finish number from 7 gives the topological number of Figure 10.1.5.

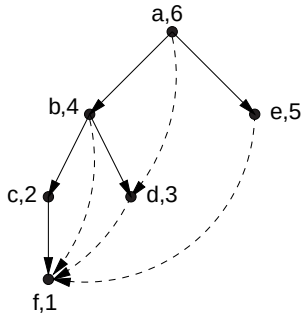

Figure 10.1.6 Topologically numbering by finish times.

F26: Another linear-time topological numbering algorithm [Kn73] works by repeatedly deleting a source. The algorithm maintains a queue of sources, as well as the in-degree of each vertex. If the in-degrees are not initially available this algorithm can do more work than Algorithm 10.1.4, since it makes two passes over the graph.

F27: Dag algorithms often propagate information from higher topological numbers to lower, either after scanning each edge (v, w) or at the end of `DFS`(v). Propagating information in the opposite direction is also possible.

F28: As an example suppose each edge e of a dag G has a real-valued length $\ell[e]$. We can find the longest path in G in linear time. The idea is to set $d[v]$ to the length of a longest path starting at v. These values $d[v]$ can be computed in reverse topological order, using the recurrence

$$d[v] = \max\{0,\ \ell[v,w] + d[w] : (v,w) \in E\}$$

It is easy to modify `DFS` to calculate these values.

The algorithm can recover the longest path from the $d[\,]$ values in a second pass. The second pass can be faster if the first pass stores a pointer for each vertex indicating its successor on its longest path. Longest paths are useful in critical path scheduling. Finding the longest path in a general graph is NP-complete.

F29: Similar algorithms can be used to calculate the longest path from s to t, shortest paths from a vertex s, etc.

F30: More generally Fact 28 illustrates how the technique of dynamic programming can be used to solve problems on dags. Dynamic programming is based on similar recurrences [CLRS01].

EXAMPLE

E17: Figure 10.1.7 illustrates how the algorithm finds a longest path in a dag. Edges are labelled with their length, and vertices are labelled with their $d[\,]$ values. The longest path corresponds to the largest $d[\,]$ value, which is 5; it is the upper path from source to sink.

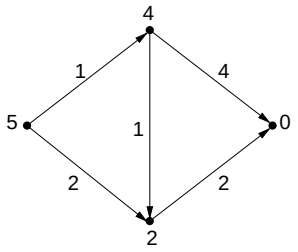

Figure 10.1.7 Longest path algorithm.

10.1.4 Connectivity Properties

Depth-first search is the method of choice to calculate low order connectivity information. This section surveys notions of 1- and 2- connectivity. It starts with 1-connectivity of directed graphs, and then examines 2-connectivity of undirected graphs. These connectivity algorithms are originally due to Tarjan [Ta72]. This section follows the path-based development of [Ga00], which simplifies the algorithms to eliminate the depth-first spanning tree.

Strong Components of a Directed Graph

In this section, $G = (V, E)$ is a directed graph.

DEFINITIONS

D20: For two vertices u and v, a ***uv*-path** is a path starting at u and ending at v.

D21: A directed graph $G = (V, E)$ is ***strongly connected*** if for every two distinct vertices u and v, there is a uv-path and a vu-path.

D22: In general, a directed graph will not be strongly connected. But the vertices can be partitioned into blocks that are strongly connected, according to this definition: two vertices u & v are in the same ***strong component (SC)*** if and only if they can reach each other, i.e., there is a uv-path and a vu-path. This defines a partition of V since it is an equivalence relation.

D23: For any directed graph, contracting each SC to a vertex gives the ***strong component graph (SC graph)***.

D24: A ***tournament*** is a directed graph G such that each pair of vertices is joined by exactly one edge. This models a round robin tournament, where edge (x, y) represents the fact that player x beat player y.

FACTS

F31: Let C be a cycle in a graph G. All vertices of C are in the same SC. Contracting the vertices of cycle C to a single vertex yields a graph with the same SC graph as G.

F32: The SC graph is always a dag. This follows from Fact 31.

F33: A topological numbering of the SC graph of a tournament gives a ranking of the players. To see why, note that if player x is in an SC with lower topological number than y, then the tournament contains the edge (x, y) (not (y, x)). Thus SC number 1 contains the players that are unequivocally in the top tier – they all beat all other players. SC number 2 contains the 2nd tier players – they all beat all other players except those in tier 1, etc.

F34: All the vertices on a cycle belong to the same SC. In fact the SC graph is formed by repeatedly contracting cycles, until no cycle remains.

F35: A sink s is a vertex of the SC graph. In fact the SC's are $\{s\}$ and the SC's of $G - s$.

F36: Facts 34–35 justify the following high-level algorithm for finding the SC graph. It repeatedly contracts a cycle or deletes a sink.

F37: Next we present a linear-time depth-first search algorithm for finding the strong components and the SC graph of a given directed graph.

Algorithm 10.1.5: Strong Components

Input: directed graph $G = (V, E)$
Output: strong components of G

repeat until G has no vertices:
    grow a dfs path P until a sink or a cycle is found
        sink s: mark $\{s\}$ as an SC & delete s from P & G
        cycle C: contract the vertices of C

Like Algorithm 10.1.3, for efficiency each iteration grows P by starting with the previous P and extending it, if possible.

F38: The algorithm has a low-level implementation that finds the SC graph in linear time [Ga00]. Sinks are deleted similar to Algorithm 10.1.3. Cycles are contracted using

a stack to represent P and another stack to give the boundaries of contracted vertices in P.

F39: The algorithm discovers each SC as a sink of the SC graph. So the SC's can be numbered in topological order by the method of Algorithm 10.1.3.

F40: The first linear-time algorithm for strong components is due to Tarjan [Ta72]. It computes a value called $lowpoint(v)$ for each vertex v. $lowpoint(v)$ is the lowest-numbered vertex (in preorder) in v's SC that is reachable from v by a path of (0 or more) tree edges followed by a back or cross edge ($lowpoint(v)$ equals v if no smaller numbered vertex can be reached). The vertices with $lowpoint(v) = v$ are the "roots" of the strong components.

F41: A third linear-time strong component algorithm is due to Sharir [Sh81] and Kosaraju (unpublished; see also [CLRS01]). It does a depth-first search, followed by a second depth-first search on the reverse graph. This makes good sense – the first search discovers which vertices can reach which others, and the second search discovers which vertices can be reached by which others.

EXAMPLES

E18: Figure 10.1.8 shows a directed graph, its three strong components, and its SC graph. Each strong component is strongly connected.

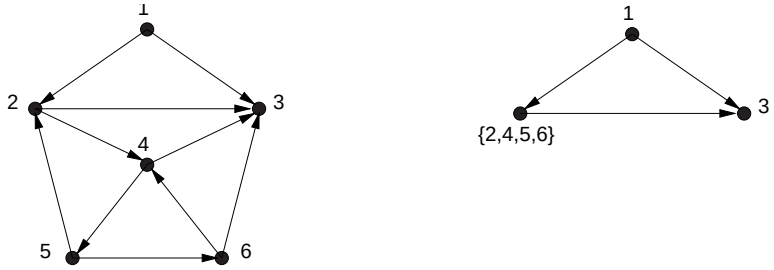

Figure 10.1.8 Strong components of a directed graph.

An elementary misperception is that a strongly connected graph has a Hamiltonian cycle. The component $\{2, 4, 5, 6\}$ illustrates that this is not always true.

E19: Figure 10.1.9 gives a dfs tree of Figure 10.1.8. (To better illustrate the algorithm a different dfs from Figure 10.1.8 is used.) Each vertex is labelled by its preorder number followed by its lowpoint value.

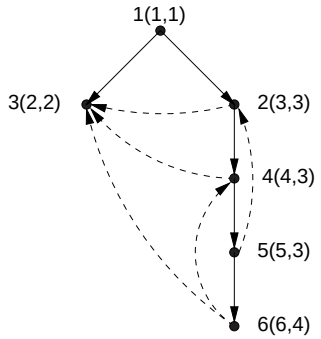

Figure 10.1.9 Execution of strong component algorithm.

E20: Suppose we number the vertices of an arbitrary directed graph by topologically numbering the SC graph, and then listing first the vertices in SC number 1, then the vertices in SC number 2, etc. The adjacency matrix of the graph with new vertex numbers is upper block triangular. This is because no edge goes from a higher numbered SC to a lower numbered SC. For instance Figure 10.1.9 gives the adjacency matrix below. It is upper triangular except for the block corresponding to SC $\{b, d, e\}$.

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

Figure 10.1.10 Upper block triangular adjacency matrix.

E21: Example 20 shows how the SC graph is used to speed up operations on sparse matrices like Gaussian elimination, matrix inversion, finding eigenvalues, etc. The given matrix M is interpreted as a directed graph, with m_{ij} corresponding to edge (i, j). The adjacency matrix of Example 20 is constructed, and the 1 for each edge (i, j) is replaced by the value m_{ij}. The resulting block upper triangular matrix has less fill-in for Gaussian elimination and nice properties for other matrix operations [Ha69].

E22: Figure 10.1.11 below illustrates the execution of the algorithm on the graph of Figure 10.1.8.

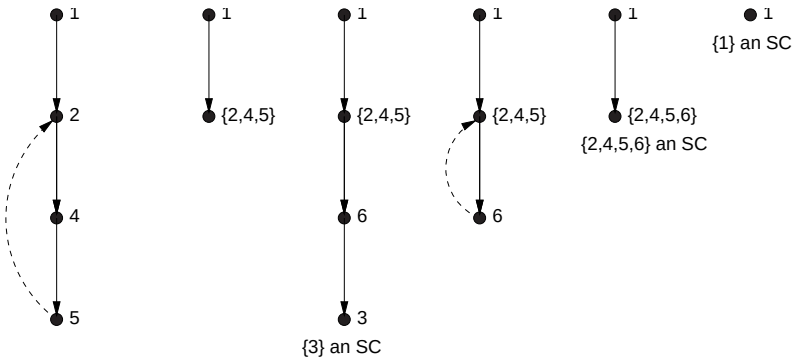

Figure 10.1.11 Execution of strong component algorithm.

E23: Figure 10.1.12 below shows a tournament and its SC graph. Player a is first, players b, d, e are in the 2nd tier, and player c is last.

E24: A Markov chain is *irreducible* if the graph of its (nonzero) transition probabilities is strongly connected.

REMARK

R7: The algorithm of Fact 41 is very simple to code and is covered in many textbooks. It can be appreciably slower than the other two algorithms, because it makes two passes over the graph and has larger memory requirement.

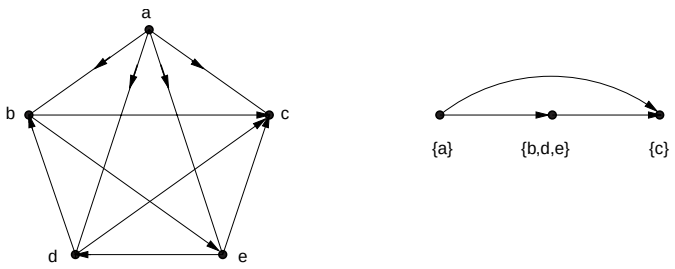

Figure 10.1.12 **Tournament and its SC graph.**

Bridges and Cutpoints of an Undirected Graph

In this section $G = (V, E)$ is a connected undirected graph.

DEFINITIONS

D25: A vertex v is an ***cutpoint (articulation point)*** if $G - v$ is not connected. A graph is ***biconnected*** if it has no cutpoint.

D26: A ***biconnected component*** is a maximal subgraph that has no cutpoint.

D27: An edge e is a ***bridge*** if $G - e$ is not connected. An edge is a bridge if and only if it's not in any cycle. A graph is ***bridgeless*** if it has no bridges.

D28: Let B be the set of all bridges of G. The ***bridge components (BCs)*** of G are the connected components of $G - B$. Equivalently a BC is the induced subgraph on a maximal set of vertices, any of which can reach any other without crossing a bridge.

D29: Contracting each BC to a vertex gives a tree, the ***bridge tree***.

D30: An ***orientation*** of an undirected graph assigns a unique direction to each edge.

D31: A ***perfect matching*** of an undirected graph G is a spanning subgraph in which every vertex has degree exactly 1.

TWO EXAMPLES

E25: Figure 10.1.13 shows a graph with 3 bridges, 6 cutpoints, and 7 biconnected components. It illustrates that an end of a bridge is a cutpoint unless it has degree one. However, a cutpoint need not be the end of a bridge.

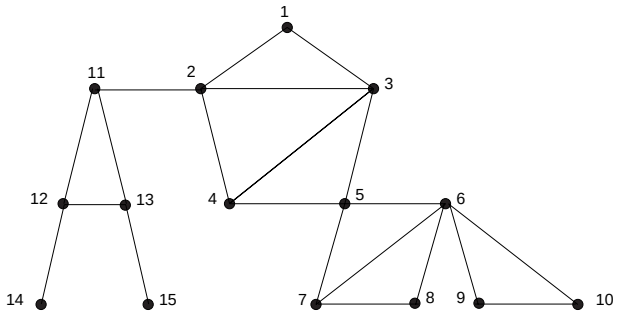

Figure 10.1.13 **Undirected graph with bridges and cutpoints.**

E26: If a communications network (e.g., Internet) has a bridge, that link's failure disables communication, i.e., there are sites that cannot send messages to each other. If the network has an articulation point, that site's failure also disables communication.

FACTS

F42: All vertices on a cycle are in the same BC. In fact the bridge tree is formed by repeatedly contracting cycles.

F43: A vertex x of degree ≤ 1 is a vertex of the bridge tree. In fact the BC's are $\{x\}$ and the BC's of $G - x$.

F44: Facts 42 and 43 justify the following high level algorithm for finding the bridges and bridge tree. It has a linear-time implementation almost identical to Algorithm 10.1.5, the strong component algorithm. We call the last vertex x of a dfs path a ***dead end*** if x has degree ≤ 1.

Algorithm 10.1.6: Bridges

Input: connected undirected graph $G = (V, E)$
Output: bridge components and bridges of G

repeat until G has no vertices:
    grow a dfs path P until a cycle is found or a dead end is reached
        cycle C: contract the vertices of C
        dead end x: mark $\{x\}$ as a BC
                if x has degree 1, then mark its edge as a bridge (of G)

F45: A similar linear-time algorithm finds the cutpoints and biconnected components of an undirected graph [Ga00].

F46: The original linear-time dfs algorithm of Hopcroft and Tarjan for cutpoints and biconnected components [Ta72] is based on the idea of lowpoints (recall Fact 40).

Start with a dfs tree T. Assume that the vertices are numbered in discovery order and that each vertex is identified with its discovery number. Define

$$lowpoint(v) = \min\{v\} \cup \{w : \text{some back edge goes from a descendant of } v \text{ to } w\}$$

Hopcroft and Tarjan proved that G is biconnected if and only if

(i) vertex 1 has exactly one child (which must be vertex 2);

(ii) $lowpoint(2) = 1$;

(iii) each vertex $w > 2$ has $lowpoint(w) < v$, where v is the parent of w.

The cutpoints have a similar characterization.

Lowpoint is easy to compute in a bottom-up pass over T, since

$$lowpoint(v) = \min\{v\} \cup \{lowpoint(w) : w \text{ a child of } v\} \cup \{w : (v, w) \text{ a back edge}\}$$

MORE EXAMPLES

E27: Figure 10.1.14 below illustrates the execution of the bridge algorithm on the graph of Figure 10.1.13.

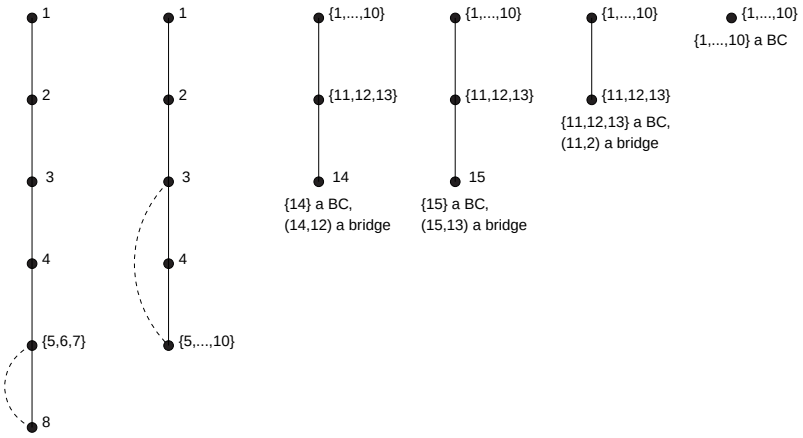

Figure 10.1.14 Execution of bridge algorithm.

E28: Figure 10.1.15 below illustrates **Robbins's Theorem** that a connected undirected graph has a strongly connected orientation if and only if it is bridgeless [Ro39]. If one of the horizontal edges is deleted, making the other a bridge, then the graph has no strongly connected orientation.

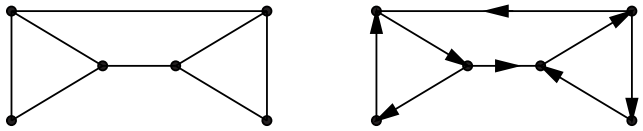

Figure 10.1.15 Undirected graph and strongly connected orientation.

E29: **Kotzig's Theorem** [Ko79] states that a unique perfect matching must contain a bridge of G. Figure 10.1.16 shows a graph with a unique perfect matching – matched edges are drawn heavy. Note that deleting the bridge of the matching gives another graph with a unique perfect matching. This idea can be used to efficiently find a unique perfect matching or show it does not exist [GaKaTa01].

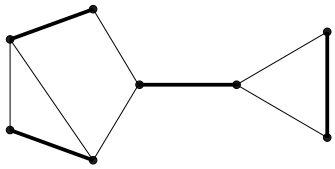

Figure 10.1.16 Graph with a unique perfect matching.

E30: **Whitney's Flipping Theorem** asserts that a graph is planar if and only if each biconnected component is planar [Wh32a].

10.1.5 DFS as a Proof Technique

In addition to being a powerful algorithmic tool, depth-first search can be used to easily prove many theorems of graph theory. (It's a handy way to remember the theorems too!) This subsection gives several examples.

DEFINITIONS

D32: A ***mixed graph*** G can have both directed and undirected edges.

D33: A mixed graph G is ***traversable*** if every ordered pair of vertices u, v has a uv-path with all its directed edges pointing in the forward direction. (Traversability is equivalent to connectedness if G is undirected and to strong connectedness if G is directed.)

D34: A ***bridge*** in a mixed graph is an undirected edge that is a bridge of G when edge directions are ignored.

D35: An ***orientation*** of a mixed graph assigns a unique direction to each undirected edge.

EXAMPLES

E31: Robbins's Theorem can be proved using the high-level bridge algorithm (Algorithm 10.1.6) and the strong components algorithm (Algorithm 10.1.5). When the BC algorithm is executed on a bridgeless graph G, it ends with G contracted to a single vertex. But if the SC algorithm ends with the entire graph contracted to a single vertex, then the initial graph is strongly connected. So orient the given undirected graph G to make the execution of the SC algorithm on the orientation mimic the execution of the BC algorithm on G. To do this orient edges that extend the dfs path or cause contractions (in the BC algorithm) so they do the same in the SC algorithm.

This is illustrated in Figure 10.1.17, which shows how a depth-first search executed on the undirected graph of Figure 10.1.15 gives the orientation shown in that figure. Enlarged hollow vertices are contractions of original vertices.

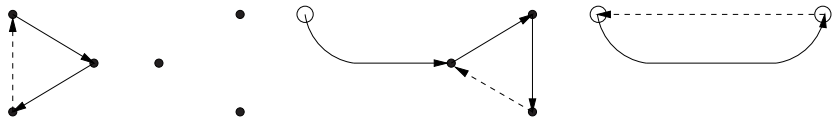

Figure 10.1.17 Dfs proof of Robbins's Theorem.

E32: The same approach proves a generalization of Robbins's theorem by Boesch and Tindell [BoTi80] that a traversable graph has a strongly-connected orientation if and only if it has no bridge. It can be proved using Algorithm 10.1.5, with the sink rule replaced by a rule for a "1-sink", i.e., a vertex with no leaving directed edge and only one incident undirected edge.

E33: Kotzig's Theorem can be proved by dfs [Ga79]. We illustrate by proving a simple special case: a bipartite graph with a unique perfect matching has a vertex of degree one. The idea is to grow a dfs path P two edges at a time, repeatedly adding an unmatched edge (x, y) and the matched edge containing y. When the path cannot be extended the last vertex y has degree 1. If not a back edge from y creates an even length cycle, whose edges yield another perfect matching as shown in Figure 10.1.18 below.

A linear-time dfs algorithm for testing if a perfect matching is unique is given in [GaKaTa01].

E34: ***Rédei's Theorem*** [Re34] states that any tournament has a Hamiltonian path, i.e., a simple path through all its vertices. This is easy to see by dfs: listing the vertices in order of decreasing finish time gives a Hamiltonian path.

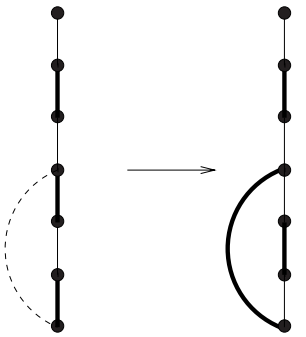

Figure 10.1.18 Dfs proof of Kotzig's Theorem.

10.1.6 More Graph Properties

The basic properties of depth-first search were developed by Hopcroft and Tarjan as stepping-stones to their goal of an efficient planarity algorithm. This subsection starts by surveying the high-level principles of the planarity algorithm. It then surveys other important properties that can be decided by efficient dfs algorithms. The depth-first tree plays a central role in all these algorithms.

Planarity Testing

The first complete linear-time algorithm to decide whether or not a graph is planar is due to Hopcroft and Tarjan. This property has obvious applications to graph drawing, circuit layout, etc. This section gives the high-level depth-first approach.

DEFINITIONS

D36: Let G be a biconnected graph with a cycle C. The edge set $E - E(C)$ can be partitioned into a family of subgraphs called ***segments*** as follows:

(i) An edge not in C that joins 2 vertices of C is a segment.

(ii) The remaining segments each consist of a connected component of $G - V(C)$, plus all edges joining that component to C.

D37: Two segments S, T of a cycle C in a graph ***interlace*** either if $|V(S) \cap V(T) \cap V(C)| \geq 3$, or if there are 4 distinct vertices u, v, w, x that occur along cycle C (not necessarily consecutively) in that order such that $u, w \in S$ and $v, x \in T$.

EXAMPLE

E35: Figure 10.1.19 below shows a cycle C (dotted) with 5 segments. Segments S_1 and S_2 interlace, and S_4 interlaces with both S_3 and S_5.

FACTS

F47: By Whitney's Flipping Theorem (Example 30), one can test planarity by treating each biconnected component separately.

F48: The graph theoretic approach used by Hopcroft and Tarjan is the following

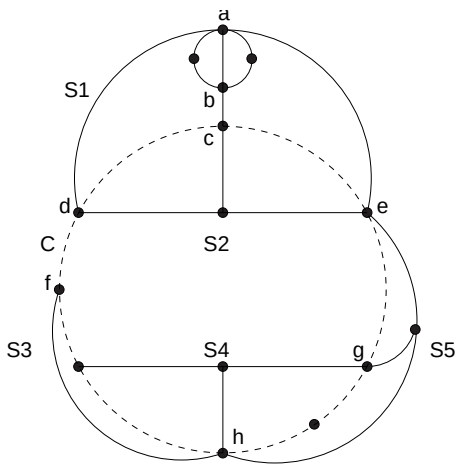

Figure 10.1.19 Planar graph with interlacing segments.

theorem of Auslander and Parter [AuPa61]: a biconnected graph G is planar if and only if

(a) $C \cup S$ is planar for every segment S;

(b) the segments can be partitioned into two families such that no two segments in the same family interlace.

The necessity of both (a) and (b) is clear. An outline of a complete proof of this theorem is given in [Ev79].

F49: Here is the overall structure of the algorithm of Hopcroft and Tarjan [HoTa74] which decides in linear time whether or not a graph is planar. Each biconnected component is processed separately.

A depth-first spanning tree of the component is found.

A cycle C is chosen, consisting of a path in the dfs tree plus one back edge.

Then segments are found:

(i) each back edges that joins two vertices of C is a segment;

(ii) each remaining segment S is determined by a vertex $w \notin C$ whose parent is in C. The edges of S are those edges with at least one endpoint descending from w. (Specifically, this amounts to the tree edge joining w to its parent, plus all edges of the subtree rooted at w, plus all back edges that join two descendants of w or join a descendant of w with a vertex of C.)

The algorithm processes each segment S recursively, checking that $C \cup S$ is planar and S can be added to an imbedding of all subgraphs processed so far. (The latter uses the interlacing criterion.)

F50: A number of additional ideas are used to achieve linear time. The *lowpoint* values (Fact 46) are used to guide the construction of cycles C. In fact the "second lowpoint" is also used. A second depth-first search is done for cycle generation. The planarity algorithm is intricate, but is very fast in practice.

Triconnectivity

Hopcroft and Tarjan show how to find the triconnected components in linear time [HoTa73b]. Like their planarity algorithm the approach is based on segments.

DEFINITIONS

D38: An undirected graph is ***triconnected*** if it is connected and remains so whenever any two or fewer vertices are deleted.

D39: Two vertices in a biconnected graph form a ***separation pair*** if deleting them leaves a disconnected graph.

There is a natural definition of the *triconnected components* of a graph.

EXAMPLE

E36: In Figure 10.1.19 above there are 5 separation pairs: a, b; a, c; d, e; e, f; and g, h.

FACTS

F51: The following characterization of the separation pairs is easy to prove. Let G be a biconnected graph with a cycle C. Let a, b be a separation pair. Then a and b either both belong to C or both belong to a common segment. Moreover, suppose a and b both belong to C. Then either

(a) some segment S has $V(S) \cap V(C) = \{a, b\} \subset V(S)$; or

(b) $C - \{a, b\}$ consists of two nonempty paths, and no segment contains a vertex of both paths.

(The symbol "$\subset$" denotes proper set containment.)

F52: The triconnectivity algorithm applies the characterization of Fact 51 recursively. Hopcroft and Tarjan's triconnectivity algorithm shares algorithmic ideas with their planarity algorithm.

F53: Another useful fact is that the two vertices of a separation pair are related (Definition 10).

Ear Decomposition and st-numbering

DEFINITIONS

D40: An ***open ear decomposition*** of an undirected graph is a partition of the edges into a simple cycle P_0 and simple paths $P_1, \ldots, P_k$ such that for each $i > 0$, P_i is joined to previous paths only at its (2 distinct) ends, i.e., $V(P_i) \cap V(\cup_{j<i} P_j)$ consists of the 2 ends of P_i. (The concept, but not the name, is due to Whitney.)

D41: Let (s, t) be any edge of a biconnected graph. An ***st-numbering*** numbers the vertices from 1 to n so that s is numbered 1, t is numbered n, and every other vertex has both a higher-numbered neighbor and a lower-numbered neighbor.

EXAMPLE

E37: Figure 10.1.20 shows an ear decomposition consisting of cycle P_0 and simple paths $P_1, \ldots, P_6$. The 15 vertices are numbered in an st-numbering (corresponding to the ear decomposition).

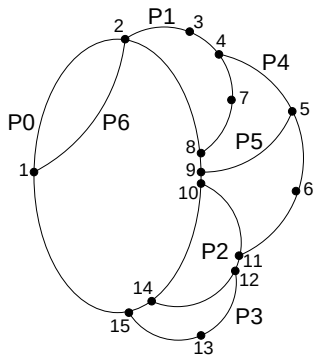

Figure 10.1.20 **Ear decomposition and st-numbering of a biconnected graph.**

FACTS

F54: Whitney [Wh32b] proved that an undirected graph is biconnected if and only if it has an open ear decomposition.

F55: An algorithm based on *lowpoint* values can be used to find an open ear decomposition of a biconnected graph in linear time (*pathfinder* in [EvTa76], although the term "ear decomposition" is not used).

F56: An open ear decomposition with $(s,t) \in P_0$ can be used to give an st-numbering in linear time [EvTa76].

REMARK

R8: st-numbering is the basis of the linear-time planarity algorithm of Lempel, Even and Cederbaum [LeEvCe67]. It constructs a planar imbedding by repeatedly adding a vertex. More precisely it starts with an imbedding of one vertex and its incident edges. Then it repeatedly adds all edges incident to the next vertex, updating the imbedding. The vertices are added in st-order.

R9: Ear decomposition is closely related to depth-first search. An open ear decomposition can be found efficiently on parallel computers with large numbers of processors; the same cannot be said for doing a depth-first search. Efficient parallel algorithms for bi- and triconnectivity and planarity are based on ear decomposition [Ra93].

Reducibility

DEFINITIONS

D42: A ***(program) flow graph*** is a directed graph with a distinguished vertex r, the ***start vertex***, that can reach every vertex.

D43: A flow graph is ***reducible*** if it can be transformed into the single vertex r by a sequence of operations of the following type:

> if (v, w) is the only edge entering w and if $w \neq r$, then contract edge (v, w) to vertex v.

(The contraction operation discards parallel edges and self-loops.)

D44: We define a problem in data structures that arises in many dfs algorithms (and other contexts). A universe of n elements is given. The problem is to maintain a partition $\mathcal{P}$ of this universe into sets.

> Initially each element forms a singleton set of $\mathcal{P}$.
>
> Partition $\mathcal{P}$ is updated by the operation $union(A, B)$, which replaces two sets A and B of $\mathcal{P}$ by their union $A \cup B$.
>
> A second operation $find(x)$ computes the name of the set currently containing element x.

The ***set-merging problem*** is to process a sequence of m intermixed *union* and *find* operations.

EXAMPLE

E38: Figure 10.1.21 shows an irreducible flow graph. In fact, this flow graph gives a forbidden subgraph characterization of reduciblity: a flow graph is reducible if and only if it does not contain a subgraph consisting of 4 vertices r, a, b, c (where r and a may coincide but otherwise the vertices are distinct) joined by vertex disjoint paths from r to a, a to b, a to c, b to c and c to b.

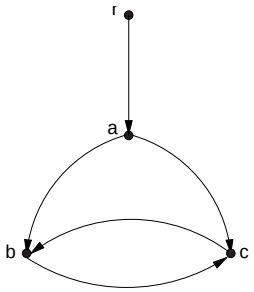

Figure 10.1.21 Irreducible flow graph.

REMARK

R10: Flow graphs model the stucture of computer programs. Any program without goto's has a reducible flow graph. Many methods for code optimization (e.g., eliminating common subexpressions, identifying active variables, finding useless definitions, etc.) depend on the graph being reducible [AhSeUl86].

FACTS

The starting point of a linear-time reducibility algorithm of Tarjan [Ta74a] for flow graphs is a reformulation of reducibility. It produces the sequence of contractions that reduce it to the start vertex. For each vertex w in a dfs tree of a flow graph, we define

$$I(w) = \{v : \text{there is a simple } vw\text{-path ending with a back edge (to } w)\}$$

F57: [Ta74a] A flow graph is reducible if and only if every set $I(w)$ consists only of descendants of w.

F58: [Ta74a] Assume that the vertices of the flow graph are indexed by preorder number. Let w be the largest vertex (in preorder) with an entering back edge. Suppose that $I(w)$ consists only of descendants of w. If we contract the vertices of $I(w)$ into vertex w, then the new graph is reducible if and only if the original was.

F59: Fact 58 specifies a sequence of contractions that substantiate the reducibility of a graph. The sets $I(w)$ can be computed simply by scanning edges in the backwards direction, starting at w. If a nondescendant of w is ever reached, then the graph is not reducible. The efficient descendance test of Fact 16 is used.

F60: The contractions performed by the algorithm change the vertex set of the graph. At all times the vertices of the current graph form a partition of the original vertex set. This partition is manipulated by the *union* and *find* operations (Definition 44).

F61: The best known algorithm for set-merging is based on the so-called weighted union and path compression rules. Tarjan showed that this algorithm solves the set-merging problem in time $O(m\alpha(m,n))$. Here α is an inverse of Ackermann's function and is very slowly growing [Ta75, CLRS01].

F62: Gabow and Tarjan [GaTa85] showed that a special case of the set-merging problem can be solved in linear time. Using this special case algorithm makes the reducibility algorithm run in linear time.

F63: Suppose the vertices are numbered in discovery order. If $v < w$ then any vw-path contains a common ancestor of v and w [Ta72]. This property holds in both directed and undirected graphs.

EXAMPLE

E39: Figure 10.1.22 shows a depth-first spanning tree of a flow graph with the vertices labelled by discovery number. Any path from vertex 5 to vertex 8 passes through one or both of the common ancestors of 5 and 8, i.e., vertices 1 and 2.

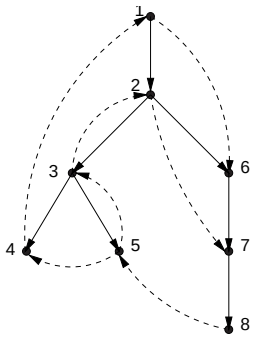

Figure 10.1.22 Depth-first search with preorder numbers.

Two Directed Spanning Trees

DEFINITIONS

D45: In a flow graph, an ***r*-tree** is a directed spanning tree rooted at start vertex r.

D46: Consider a dfs tree with root r. Edge (v, w) is a ***(directed) bridge*** in a flow graph if every rw-path includes (v, w).

EXAMPLE

E40: In Figure 10.1.21 edge (r, a) is a bridge. Duplicating it gives a graph with 2 edge-disjoint r-trees (the directed paths r, a, b, c and r, a, c, b).

FACTS

Consider a dfs tree with root r. For any vertex w define

$$I(w) = \{v : \text{there is a simple } vw\text{-path containing only descendants of } w\}$$

Clearly the path of this definition ends in a back edge to w (unless $v = w$). Note the similarity with Fact 57.

F64: A flow graph has two edge-disjoint r-trees if and only if each vertex $v \neq r$ has two edge-disjoint paths rv-paths (recall Definition 20). This is a special case of ***Edmonds's Branching Theorem*** which is the same statement generalized from 2 to any $k \geq 2$ [Ed72].

F65: A flow graph has 2 edge-disjoint r-trees if and only if there are no bridges.

F66: Tarjan presents a linear-time algorithm to find 2 edge-disjoint r-trees if they exist [Ta74a]. More generally in an arbitrary flow graph the algorithm finds 2 r-trees that contain the fewest possible number of common edges. The more general problem is solved by identifying the bridges and duplicating each of them. This gives a graph with 2 edge-disjoint r-trees (Fact 65).

F67: In terms of dfs, an edge (v, w) is a bridge if and only if (v, w) is a tree edge and is the only edge entering $I(w)$. Tarjan's algorithm identifies the bridges using techniques similar to the reducibility algorithm. Computing the trees is more involved.

F68: The algorithm performs set-merging to keep track of the contracted vertices, as in the reducibility algorithm. As in that algorithm the data structure of [GaTa85] is used to achieve linear time.

Dominators

DEFINITIONS

D47: In a flow graph with start vertex r, vertex v ***dominates*** vertex $w \neq v$ if every rw-path contains v.

D48: The ***immediate dominator*** of w, denoted $idom(w)$, is a vertex v that dominates w such that every other dominator of w dominates v.

D49: The ***dominator tree*** is a tree T whose nodes are the vertices of G. The root of T is the start vertex r. The parent of a vertex $v \neq r$ is $idom(v)$.

D50: The ***internal vertices*** of a path are all its vertices except the endpoints.

D51: Consider a dfs tree with root r. For every vertex $w \neq r$, the ***semidominator*** of w, $sdom(w)$, is defined by

$$sdom(w) = \min\{v : \text{some } vw\text{-path has all its internal vertices} > w\}$$

EXAMPLE

E41: Figure 10.1.23 shows the dominator tree for the graph of Figure 10.1.22. Note that vertex 2 does not dominate 3 because of path $1, 6, 7, 8, 5, 3$. The start vertex 1 is the semidominator of every vertex except two: $sdom(7) = 2$, $sdom(8) = 7$. Although vertex 1 is the immediate dominator of 7 it is not the semidominator of 7.

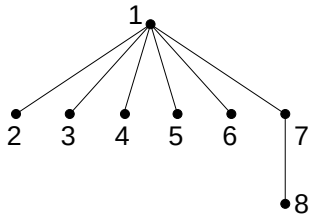

Figure 10.1.23 Dominator tree for Figure 10.1.22.

FACTS

F69: The basic properties of dominance are due to Lowry and Medlock [LoMe69]: Every vertex except r has a unique immediate dominator. This justifies the notion of dominator tree. A vertex v dominates w if and only if v is a proper ancestor of w in the dominator tree.

F70: Lengauer and Tarjan [LeTa79] give an efficient algorithm to find the dominator tree T. It is a refinement of an earlier dfs algorithm of Tarjan [Ta74b].

F71: For any vertex w, $sdom(w)$ is a proper ancestor of w. This follows from Fact 63.

Semidominators are useful because of the next two facts proved by Lengauer and Tarjan:

F72: Take any vertex $w \neq r$. Let u be a vertex with minimum value $sdom(u)$ among all vertices in the tree path from $sdom(w)$ to w, excluding $sdom(w)$. Then

$$idom(w) = \begin{cases} sdom(w) & \text{if } sdom(w) = sdom(u) \\ idom(u) & \text{otherwise} \end{cases}$$

F73: Semidominators can be computed by a recursive definition:

$$sdom(w) = \min\{v : (v, w) \text{ an edge}\} \cup \{sdom(u) : u > w \text{ and some edge goes from a descendant of } u \text{ to } w\}$$

(Note the similarity with *lowpoint* in Fact 46.)

F74: The algorithm of Lengauer and Tarjan [LeTa79] computes semidominators using Fact 73 in a backwards pass (i.e., w is decreasing). Then it computes immediate dominators using Fact 72 in a forwards pass.

F75: The time for the algorithm is $O(m\alpha(m, n))$. An implementation of this algorithm in linear time is presented in [AlHaLaTh99].

10.1.7 Approximation Algorithms

Finding small spanning subgraphs with prespecified connectivity properties is usually a difficult (NP-hard) problem. For example, finding a bridgeless spanning subgraph

with the fewest possible number of edges is NP-hard. (The reason is that this subgraph contains n edges if and only if there is a Hamiltonian cycle.)

Depth-first search has been used to design good approximation algorithms for such difficult problems. Here the goal is to find a subgraph that has all the desired properties except that instead of having the fewest possible number of edges, it is within a small constant factor of this goal. This section surveys the use of depth-first search in approximation algorithms for connectivity properties. Other dfs approximation algorithms are surveyed in [Kh97].

DEFINITIONS

D52: Consider an optimization problem that seeks to find a smallest feasible solution OPT. An ***α-approximation algorithm*** is a polynomial-time algorithm that is guaranteed to find a solution of size at most $\alpha|OPT|$ [CLRS01]. For the graph problems of this section, the size of the solution is the number of edges.

D53: The ***smallest bridgeless spanning subgraph*** of a connected bridgeless undirected graph is a bridgeless spanning subgraph with the minimum possible number of edges.

D54: An undirected graph is ***k-edge connected*** if it is connected and remains so when any set of fewer than k edges is deleted. This concept makes good sense for a multigraph. A ***k-ECSS*** is a k-edge connected spanning subgraph; the graph is assumed to be k-edge connected. So a bridgeless spanning subgraph is a 2-ECSS. A ***smallest k-ECSS*** has the fewest possible number of edges. From now on instead of "smallest bridgeless spanning subgraph", we use the shorter equivalent phrasing, "smallest 2-ECSS".

ALGORITHM

Approximation algorithms for the smallest 2-ECSS are our first concern. A 2-approximation can be designed from Algorithm 10.1.6 in a straightforward way. Khuller and Vishkin [KhVi94] were the first to go beyond this. They presented an elegant dfs algorithm based on a "tree carving" using the dfs tree. The following modification of Algorithm 10.1.6 is a path-based reinterpretation of their algorithm.

Algorithm 10.1.7: Smallest 2-ECSS Approximation

Input: bridgeless undirected graph $G = (V, E)$
Output: edge set $F \subseteq E$, a 3/2-approximation to the smallest 2-ECSS

$F = \emptyset$
repeat until G has 1 vertex:
    grow a dfs path P until its endpoint x has all neighbors belonging to P
    let y be the neighbor of x closest to the start of P
    let C be the cycle formed by edge (x, y) & edges of P
    add all edges of C to F
    contract the vertices of C

EXAMPLE

E42: Figure 10.1.24 below gives a sample execution of the algorithm. The given graph on top has a Hamiltonian cycle, so the smallest 2-ECSS has n edges.

The algorithm grows the depth-first path of solid edges shown in the middle, starting from r. It then adds the dashed edges.

A typical edge addition is illustrated in the bottom graph, where the enlarged hollow vertex is the contraction of the last vertices on the path.

As n approaches ∞, the algorithm's solution approaches $3n/2$ edges: n solid edges and $n/2$ dashed edges. So the approxmation ratio approaches 3/2.

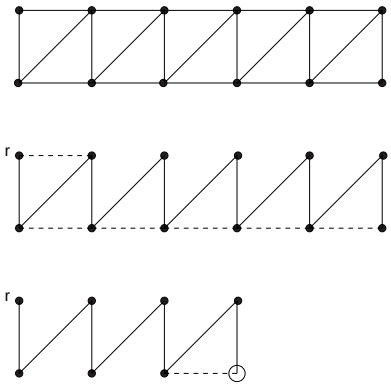

Figure 10.1.24 Smallest 2-ECSS approximation algorithm.

FACTS

Good approximation algorithms require good lower bounds on the size of the optimum solution. We analyze this algorithm using 2 lower bounds.

F76: The ***Degree Lower Bound*** says that any 2-ECSS has at least n edges. This results from the fact that every vertex must have degree at least 2.

F77: The ***Carving Lower Bound*** says that if Algorithm 10.1.7 contracts c cycles, then any 2-ECSS has at least $2c$ edges [KhVi94]. To see this let x be an endpoint of P giving a contraction in the algorithm. Any 2-ECSS contains ≥ 2 edges leaving x. These edges disappear in the contraction operation. So we can repeat this argument for every contraction, getting a lower bound of $2c$ edges.

F78: Algorithm 10.1.7 is a 3/2 approximation. This follows because the edge set F consists of $n-1$ edges from paths P and c edges that cause contractions. If OPT is the edge set of a 2-ECSS, then $|OPT| > n$ (Degree Lower Bound) and $|OPT|/2 \geq c$ (Carving Lower Bound). Thus $|F| < 3|OPT|/2$.

F79: Vempala and Vetta [VeVe00] present a 4/3-approximation algorithm for the smallest 2-ECSS. Their algorithm is based on the idea of doing a depth-first search of objects of the graph, specifically cycles and paths. It uses the ***Matching Lower Bound***: Any 2-ECSS has at least as many edges as a smallest spanning subgraph where every vertex has degree ≥ 2. Vempala and Vetta give a similar 4/3-approximation algorithm for the smallest biconnected subgraph of a biconnected graph.

F80: Jothi, Raghavachari & Varadrajan [JoRaVa03] use a stronger version of the Matching Lower Bound in a dfs algorithm that achieves performance ratio 5/4 for the smallest 2-ECSS. Vetta [Ve01] uses a version of the Matching Lower Bound in a dfs algorithm that approximates the smallest strongly connected subgraph of a strongly connected graph to within a factor 3/2.

F81: The Carving Lower Bound extends to k-ECSS: If Algorithm 10.1.7 contracts c cycles then any k-ECSS has at least kc edges [KhVi94]. This can be proved by simply changing the "2"'s to k's in Fact 77.

F82: Gabow [Ga02] gives a dfs algorithm that is a 3/2-approximation for the smallest 3-ECSS of a multigraph. It uses the above dfs approach of [KhVi94] for 2-ECSS, the Carving Lower Bound, and ear decomposition.

F83: Khuller and Raghavachari [KhRa96] present the first approximation algorithm that achieves ratio < 2 for the smallest k-ECSS of a multigraph. It boosts the edge-connectivity of the solution graph in steps of 2. Each of these steps is a slight variant of the above algorithm of [KhVi94]. The analysis is based on a refinement of the Carving Lower Bound. Gabow [Ga03] improves the analysis to show it is a 1.61-approximation.

REFERENCES

[AhSeUl86] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.

[AlHaLaTh99] S. Alstrup, D. Harel, P. W. Lauridsen and M. Thorup, Dominators in linear time, *SIAM J. Comput.* 28 (1999), 2117–2132.

[AuPa61] L. Auslander and S. V. Parter, On imbedding graphs in the plane, *J. Math. and Mech.* 10 (1961), 517–523.

[BoTi80] F. Boesch and R. Tindell, Robbins's theorem for mixed multigraphs, *Amer. Math. Monthly* 87 (1980), 716–719.

[BrBr96] G. Brassard and P. Bratley, *Fundamentals of Algorithmics*, Prentice-Hall, 1996.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, 2nd Ed., McGraw-Hill, 2001.

[Ed72] J. Edmonds, Edge-disjoint branchings, pp. 91–96 in *Combinatorial Algorithms*, R. Rustin, Ed., Algorithmics Press, New York (1972).

[Ev79] S. Even, *Graph Algorithms*, Computer Sci. Press, MD, 1979.

[EvTa76] S. Even and R. E. Tarjan, Computing an st-numbering, *Theoret. Comp. Sci.* 2 (1976), 339–344.

[FrTa87] M. L. Fredman and R. E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, *J. ACM* 34 (1987), 596–615.

[Ga79] H. N. Gabow, Algorithmic proofs of two relations between connectivity and the 1-factors of a graph, *Disc. Math.* 26 (1979), 33–40.

[Ga00] H. N. Gabow, Path-based depth-first search for strong and biconnected components, *Inf. Proc. Letters* 74 (2000), 107–114.

[Ga02] H. N. Gabow, An ear decomposition approach to approximating the smallest 3-edge connected spanning subgraph of a multigraph, *Proc. 13th Annual ACM-SIAM Symp. on Disc. Algorithms* (2002), 84–93.

[Ga03] H. N. Gabow, Better performance bounds for finding the smallest k-edge connected spanning subgraph of a multigraph, *Proc. 14th Annual ACM-SIAM Symp. on Disc. Algorithms* (2003), 460–469.

[GaTa85] H. N. Gabow and R. E. Tarjan, A linear-time algorithm for a special case of disjoint set union, *J. Comp. and Sys. Sci.* 30 (1985), 209–221.

[GaKaTa01] H. N. Gabow, H. Kaplan and R. E. Tarjan, Unique maximum matching algorithms, *J. Algorithms* 40 (2001), 159–183.

[GoTa02] M. T. Goodrich and R. Tamassia, *Algorithm Design: Foundations, Analysis and Internet Examples*, John Wiley & Sons, 2002.

[Ha69] F. Harary, *Graph Theory*, Addison-Wesley, Reading MA, 1969.

[HoTa73a] J. Hopcroft and R. E. Tarjan, Efficient algorithms for graph manipulation, *Comm. ACM* 16 (1973), 372–378.

[HoTa73b] J.E. Hopcroft and R.E. Tarjan, Dividing a graph into triconnected components, *SIAM J. Comput.* 2 (1973), 135–158.

[HoTa74] J. Hopcroft and R. Tarjan, Efficient planarity testing, *J. ACM* 21 (1974), 549–568.

[HSR98] E. Horowitz, S. Sahni and S. Rajasekaran, *Computer Algorithms*, Computer-Science Press, 1998.

[JoRaVa03] R. Jothi, B. Raghavachari and S. Varadarajan, A 5/4-approximation algorithm for minimum 2-edge-connectivity, *Proc. 14th Annual ACM-SIAM Symp. on Disc. Algorithms* (2003) 725–734.

[Kh97] S. Khuller, Approximation algorithms for finding highly connected subgraphs, in *Appoximation Algorithms for NP-hard Problems*, D. S. Hochbaum, Ed., PWS Publishing, 1997.

[KhRa96] S. Khuller and B. Raghavachari, Improved approximation algorithms for uniform connectivity problems, *J. Algorithms* 21 (1996), 434–450.

[KhVi94] S. Khuller and U. Vishkin, Biconnectivity approximations and graph carvings, *J. ACM* 41 (1994), 214–235.

[Kl00] J. Kleinberg, The small-world phenomenon: An algorithmic perspective, *Proc. 32nd Annual ACM Symp. on Th. Comput.* (2000), 163–170.

[Kn73] D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 2nd Ed., Addison-Wesley, 1973.

[Ko59] A. Kotzig, On the theory of finite graphs with a linear factor I, *Mat.-Fyz. Časopis Slovensk. Akad. Vied* 9 (1959), 73–91.

[LeEvCe67] A. Lempel, S. Even and I. Cederbaum, An algorithm for planarity testing of graphs, *Theory of Graphs: Int. Symp*, P. Rosenstiehl, Ed., Gordon and Breach, (1967), 215–232.

[LeTa79] T. Lengauer and R. E. Tarjan, A fast algorithm for finding dominators in a flowgraph, *ACM Trans. on Prog. Lang. and Sys.* 1 (1979), 121–141.

[LoMe69] E. S. Lowry and C. W. Medlock, Object code optimization, *C. ACM* 12 (1969), 13–21.

[Lu:1882] E. Lucas, *Récreations Mathématiques*, Paris, 1882.

[Mi67] S. Milgram, The small world problem, *Psychology Today* 1 (1967), 60–67.

[Ra93] V. Ramachandran, Parallel open ear decomposition with applications to graph biconnectivity and triconnectivity, in *Synthesis of Parallel Algorithms*, J. H. Reif, Ed., Morgan Kaufmann, 1993.

[Re34] L. Rédei, Ein kombinatorischer Satz., *Acta Litt. Sci. Szeged* 7 (1934), 39–43.

[Ro39] H. E. Robbins, A theorem on graphs, with an application to a problem in traffic control, *Amer. Math. Monthly* 46 (1939), 281–283.

[Se02] R. Sedgewick, *Algorithms in C++, Part 5: Graph Algorithms*, Addison-Wesley, 2002.

[Sh81] M. Sharir, A strong-connectivity algorithm and its application in data flow analysis, *Comp. and Math. with Applications* 7 (1981), 67–72.

[Ta72] R. E. Tarjan, Depth-first search and linear graph algorithms, *SIAM J. Comput.* 1 (1972), 146–160.

[Ta74a] R. E. Tarjan, Testing flow graph reducibility, *J. Comput. Sys. Sci.* 9 (1974), 355–365.

[Ta74b] R. E. Tarjan, Finding dominators in directed graphs, *SIAM J. Comput.* 3 (1974), 62–89.

[Ta75] R. E. Tarjan, Efficiency of a good but not linear set union algorithm, *J. ACM* 22 (1975), 215–225.

[Ta76] R. E. Tarjan, Edge-disjoint spanning trees and depth-first search, *Acta Inf.* 6 (1976), 171–185.

[Tarr:1895] G. Tarry, Le probléme des labyrinthes, *Nouvelles Ann. de Math.* 14 (1895), 187.

[VeVe00] S. Vempala and A. Vetta, Factor 4/3 approximations for minimum 2-connected subgraphs, in *Appoximation Algorithms for Combinatorial Optimization*, K. Jansen and S. Khuller, Eds., Springer-Verlag, Lecture Notes in Computer Science 1931 (2000), 262–273.

[Ve01] A. Vetta, Approximating the minimum strongly connected subgraph via a matching lower bound, *Proc. 12th Annual ACM-SIAM Symp. on Disc. Algorithms* (2001), 417–426.

[We99] M. A. Weiss, *Data Structures and Algorithm Analysis in C++*, Second Ed., Addison-Wesley, 1999.

[Wh32a] H. Whitney, Non-separable and planar graphs, *Trans. Amer. Math. Soc.* 34 (1932), 339–362.

[Wh32b] H. Whitney, Congruent graphs and the connectivity of graphs, *Amer. J. Math.* 54 (1932), 150–168.

GLOSSARY

α-approximation algorithm: for minimization problems, a polynomial-time algorithm that is guaranteed to find a solution of size at most α times the minimum.

ancestor: x is an ancestor of y in a tree if there is an xy-path whose edges all go from parent to child.

articulation point: a vertex whose removal disconnects an undirected graph.

back edge: (i) in an undirected dfs a back edge is a nontree edge; (ii) in a directed dfs a back edge is directed from descendant to ancestor.

biconnected component: in an undirected graph, a maximal set of edges that has no cutpoint.

biconnected graph: an undirected graph with no cutpoint.

breadth-first search (bfs): a graph search method that finds shortest paths.

breadth-first tree: an ordered tree where the children of x are the vertices discovered from x in a breadth-first search.

bridge: (i) an edge whose removal disconnects an undirected graph; (ii) an undirected edge in a mixed graph that is a bridge when directions of edges are ignored; (iii) an edge (v, w) in a flow graph that belongs to every rw-path.

bridgeless graph: an undirected graph with no bridges.

bridge component (BC): a connected component of a connected graph when all bridges are deleted.

bridge tree: the tree formed by contracting every bridge component of a connected graph.

cross edge: in a directed dfs, a nontree edge joining two unrelated vertices.

cutpoint: see articulation point.

dag: directed acyclic graph, i.e., directed graph with no cycle.

depth-first search (dfs): a graph search method that repeatedly scans an edge incident to the most recently discovered vertex that still has unscanned edges.

depth-first spanning forest: (i) in an undirected dfs, a collection of depth-first trees, one for each connected component of G; (ii) in a directed dfs, a collection of depth-first trees containing every vertex once, with all edges joining 2 trees directed from right to left.

depth-first tree (dfs tree): an ordered tree where the children of x are the vertices discovered from x in a depth-first search.

descendant: x is a descendant of y in a tree if there is an xy-path whose edges all go from child to parent.

diameter: the maximum distance between 2 vertices.

discovery (of a vertex): when a depth-first search reaches a given vertex for the first time.

discovery order: a numbering of the vertices from 1 to n in the order they are discovered in a depth-first search.

distance (from vertex u to vertex v): length of a shortest uv-path.

dominates: in a flow graph vertex v dominates vertex $w \neq v$ if every rw-path contains v.

dominator tree: a tree representing all dominance relations in a flow graph.

k-edge connected: An undirected graph is k-edge connected if it is connected and remains so when any set of $< k$ edges is deleted.

k-ECSS: a k-edge connected spanning subgraph of a k-edge connected graph.

finished: when a depth-first search leaves a given vertex for the last time.

finish time order: a numbering of the vertices from 1 to n in the order they are discovered in a depth-first search.

flow graph: a directed graph with a distinguished vertex r that can reach every vertex.

forward edge: in a directed dfs, a nontree edge going from ancestor to descendant.

immediate dominator (*idom*): in a flow graph, the immediate dominator of vertex w is a vertex v that dominates w and every other dominator of w dominates v.

interlace: two segments S, T of a cycle C interlace when either $|V(S) \cap V(T) \cap V(C)| \geq 3$ or there are 4 distinct vertices u, v, w, x that occur in C (not necessarily consecutively) in that order such that $u, w \in S$ and $v, x \in T$.

internal vertex: a vertex of a path that is not one of the endpoints.

irreducible: a Markov chain is irreducible if the graph of its (nonzero) transition probabilities is strongly connected.

left of: in an ordered tree, vertex x is to the left of vertex y if some vertex has children c and d, with c preceding d, c an ancestor of x and d an ancestor of y.

length function: an assignment of a numerical length to each edge, often nonnegatively valued.

mixed graph: a graph with both directed and undirected edges.

open ear decomposition: a partition of the edges of an undirected graph into a simple cycle P_0 and simple paths $P_1, \ldots, P_k$ such that for each $i > 0$, P_i is joined to previous paths only at its (2 distinct) ends.

orientation: assignment of a unique direction to each undirected edge.

ordered tree: a tree where the children of each vertex are linearly ordered.

uv-path: a path starting at u and ending at v.

perfect matching: a spanning subgraph of an undirected graph where every vertex has degree exactly 1.

postorder: see finish time order.

preorder: see discovery order.

proper ancestor: x is a proper ancestor of y if it is an ancestor and $x \neq y$

proper descendant: x is a proper descendant of y if it is a descendant and $x \neq y$

reducible: A flow graph is reducible if it can be transformed into the single vertex r by a sequence of operations of the following type: If (v, w) is the only edge entering w and $w \neq r$ then contract w and v into a new vertex called v.

related: two vertices in a tree are related if one is an ancestor of the other.

scan (an edge): the work done by a graph searching algorithm when it traverses an edge.

search: a methodical exploration of all vertices and edges of a graph that runs in linear time.

segment: in a biconnected graph, a segment of a cycle C is either (i) an edge not in C that joins 2 vertices of C; or (ii) a connected component formed when the vertices of C are deleted, plus all edges joining that component to C.

semidominator: a useful intermediate concept in computing dominators, defined in terms of a depth-first search tree.

separation pair: two vertices in a biconnected graph whose removal disconnects the graph.

set-merging problem: the problem of maintaining a partition of a given universe subject to a sequence of *union* and *find* operations.

shortest-path tree: a tree where the path from the root r to any vertex v is a shortest rv-path.

sink: a vertex with outdegree 0.

smallest bridgeless spanning subgraph: a bridgeless spanning subgraph of a connected bridgeless undirected graph that has the minimum possible number of edges.

smallest k-ECSS: a k-ECSS that has the minimum possible number of edges.

source: a vertex with indegree 0.

start vertex: the distinguished vertex of a flow graph.

strongly connected: a digraph is strongly connected if every vertex can reach every other vertex by a directed path.

strong component (SC): two vertices of a directed graph are in the same strong component if they can reach each other.

strong component graph (SC graph): a directed graph with every strong component contracted to a vertex.

topological numbering (topological order, topological sort): assignment of an integer to each vertex so that each edge is directed from lower number to higher number.

tournament: a directed graph where each pair of vertices is joined by exactly one edge.

traversable: a mixed graph is traversable if every vertex can reach every other, by a path with all its directed edges pointing in the forward direction.

tree edge: edge of a dfs tree.

r-tree: a directed spanning tree rooted at r.

triconnected: an undirected graph is triconnected if it is connected and remains so whenever any two or fewer vertices are deleted.