Special Topics in Directed Graphs

MAD 3105: Discrete Mathematics II
Spring 2025

Adam Rumpf*

As with out earlier initial coverage of bipartite graphs, some of the material we will cover in the course is not included in the main textbook [3] and will thus be supplemented by lecture notes. Several of the topics to be covered in Unit 2 are applications of *directed graphs*, particularly applications related to *network flows* problems. Some basic definitions related to directed graphs can be found in our earlier notes and in *Combinatorics*, by Joy Morris [5], but these notes will discuss some specialized topics not found in our main textbook.

1 Directed Graph Basics

A **directed graph** (a.k.a. **digraph**, **network**) has edges associated with *ordered* pairs of vertices, (u, v), as opposed to an **undirected graph** (a.k.a. just a **graph**), which has edges associated with unordered pairs of vertices $\{u, v\}$. Directed graphs are typically used to represent one-way relationships, and they are usually displayed by drawing edges as arrows to indicate the direction of the relationship (see Figure 1.1). The vertices of a directed graph are sometimes called **nodes**, while their directed edges are called **arcs**. The beginning of a directed edge is called its **tail** (a.k.a. **predecessor**), while the end is called its **head** (a.k.a. **successor**).

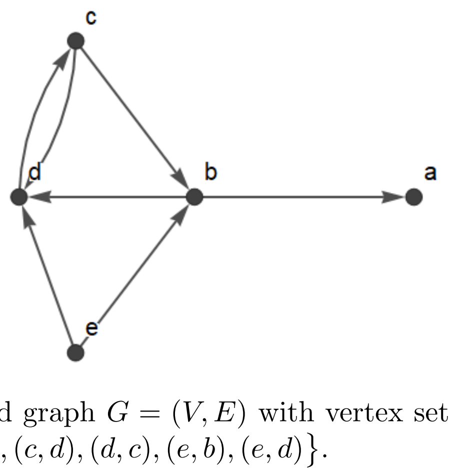

Figure 1.1: An example directed graph $G = (V, E)$ with vertex set $V = \{a, b, c, d, e\}$ and directed edge set $E = \{(b, a), (b, d), (c, b), (c, d), (d, c), (e, b), (e, d)\}$.

Because each node can have both incoming and outgoing arcs, concepts like *degree* and *neighbor* can be generalized to distinguish between the two. The **out-neighborhood** of a vertex $v \in V$ in a directed graph, denoted $N^+(v)$, is the set of successors of v. The **in-neighborhood** of v, denoted $N^-(v)$, is the set of predecessors of v. The **outdegree** of a vertex v in a directed graph,

*Florida Polytechnic University, Department of Applied Mathematics, arumpf@floridapoly.edu

denoted $d^+(v)$, is the number of directed edges that exit v, while the **indegree**, $d^-(v)$, is the number of edges that enter v. The **maximum** and **minimum** indegree and outdegree of a directed graph are defined in the obvious way as

$$\Delta^+(G) := \max_{v \in V} d^+(v) \qquad \delta^+(G) := \min_{v \in V} d^+(v)$$
$$\Delta^-(G) := \max_{v \in V} d^-(v) \qquad \delta^-(G) := \min_{v \in V} d^-(v)$$

The **underlying graph** of a directed graph is the graph obtained by converting all directed edges into undirected edges. An **orientation** of an undirected graph is a directed graph obtained by converting all undirected edges into oriented edges. Because each edge can be oriented in two ways, an undirected graph generally has many possible orientations (see Figure 1.2).

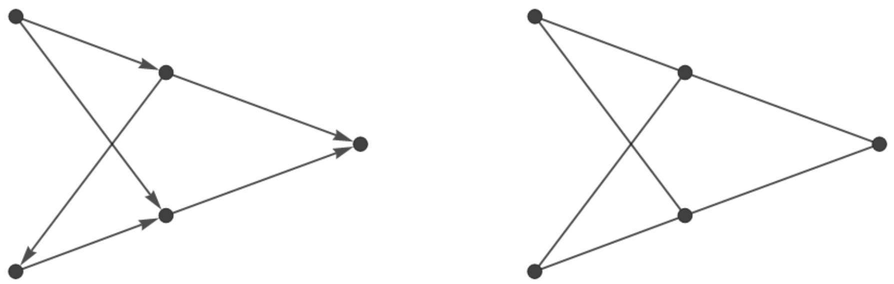

Figure 1.2: A directed graph (left) and its underlying graph (right), or equivalently an undirected graph (right) and one of its possible orientations (left).

Many results for undirected graphs generalize in a straightforward way for directed graphs. Some of these results that we've seen in class include the following.

Theorem 1.1 (Degree Sum Formula for Digraphs). The sum of indegrees of all vertices and the sum of outdegrees of all vertices in a digraph $G = (V, E)$ both equal the number of edges in G, i.e.

$$\sum_{v \in V} d^+(v) = \sum_{v \in V} d^-(v) = |E|$$

Lemma 1.2. If $\delta^+(G) \geq 1$ (or equivalently if $\delta^-(G) \geq 1$), then digraph G contains a directed cycle.

Theorem 1.3 (Eulerian Digraph Characterization). A digraph is Eulerian if and only if it has at most 1 nontrivial component and all vertices have equal indegree and outdegree.

2 Directed Acyclic Graphs

The following notes are primarily drawn from *Discrete Mathematics*, by Dossey et al. [2], with some technical details and proofs drawn from *Network Flows*, by Ahuja et al. [1].

In many applications of directed graphs, it is natural for the resulting graph model to be free of cycles. This often occurs, for example, when using a digraph to represent dependency structures or organizational charts. A **directed acyclic graph (DAG)** is a graph containing no directed cycles. Note that a DAG prohibits only *directed* cycles; its underlying graph may still contain cycles (e.g. the digraph is Figure 1.2 is a DAG).

DAGs possess many useful properties, and knowing that a directed graph is acyclic immediately implies a lot of useful results. Among other things this includes having well-defined shortest and longest paths that can both be found very efficiently. Most of these useful properties stem from the fact that DAGs can be *topologically ordered*.

2.1 Topological Orderings

A **topological ordering** of a directed graph $G = (V, E)$ is an ordering of its nodes $(v_1, v_2, \ldots, v_n)$ such that $i < j$ for each arc $(v_i, v_j) \in E$. In short, it is an ordering of the nodes such that each predecessor node appears earlier on the list than its successor node. We can see when a graph has a topological ordering by arranging its vertices in that order from left to right, in which case all arcs should point from left to right (see Figure 2.1).

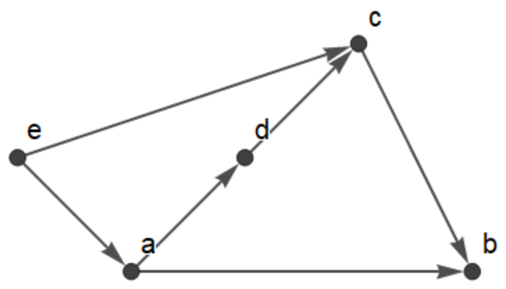

Figure 2.1: The above graph has a topological ordering (e, a, d, c, b). We can see that this is a topological ordering since each arcs points from left to right, with each predecessor earlier in the ordering than its successor.

It can be readily seen that a topological ordering cannot exist if the graph contains a directed cycle (see Figure 2.2), meaning that only DAGs have the potential to possess topological orderings. Lacking directed cycles is also *sufficient* for a digraph to have a topological ordering, as shown by the following algorithm [4].

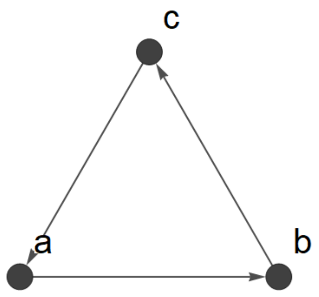

Figure 2.2: A graph with a directed cycle, which has no topological ordering. Any permutation of the nodes $\{a, b, c\}$ results in a list with at least one back-arc since the final node in the sequence must have an out-neighbor earlier in the sequence.

Topological Sorting Algorithm

```
1. Initialize $L$ as an empty list and $S$ as the set of all nodes $v \in V$ with no incoming arcs.
2. While $S \neq \emptyset$, do the following:
    (a) Remove a node $u$ from $S$ and add it to the end of $L$.
    (b) For each arc $(u, v) \in E$ exiting node $u$:
        i. Remove $(u, v)$ from $E$.
        ii. If $v$ has no more incoming arcs, add $v$ to $S$.
3. If any arcs remain in $E$, then no topological ordering exists.
   Otherwise, return topological ordering $L$.
```

It can be proved that, if G has a topological ordering, this algorithm finds it.

Proof. Firstly, note that this algorithm is guaranteed to terminate after a finite number of iterations of the main loop. Each iteration removes one node from S, and any node that exits S can never re-enter it, so the main loop can last for at most $|V|$ iterations.

To prove that L is a topological ordering, a node is only added to L after it has been in S, which only occurs after all its predecessors (if any) have already been removed from S and placed in L. Therefore no arc $(u, v) \in E$ can have u appear later than v in L. If the algorithm terminates after having processed all nodes in V, then L is a topological ordering of G.

If instead the algorithm terminates with some nontrivial component of G, say G', left over, then it must be the case that $\delta^-(G') \geq 1$ (otherwise there would be a node in G' with no predecessors, in which case that node would be in S and the algorithm would not have terminated). By Lemma 1.2, this implies that G' contains a directed cycle, and we observed earlier that directed cycles prohibit topological orderings. Therefore the algorithm terminates with edges left over if and only if no topological ordering exists. $\square$

This algorithm allows us to establish the following general result.

Proposition 2.1. $G = (V, E)$ is a directed acyclic graph if and only if there is a topological ordering of V.

The proof follows directly from the previous conclusions: if G is a DAG, then we can find a topological ordering for it (using the above algorithm), and if G is not a DAG, then it has no topological ordering.

2.2 PERT Charts and the Critical Path Method

An important application of DAGs arises in the field of project management. Suppose we wish to complete a project that requires the completion of a number of smaller tasks, each of which has its own expected time and list of requisite tasks. As a specific example, Table 2.1 shows a list of the required tasks, prerequisites, and timings for completing a particular project.

Projects of this type can be analyzed using a general method called *PERT: Program Evaluation and Review Technique.* The PERT method begins by generating a **PERT chart**, consisting of a directed graph with a node for each milestone and an arc for each tasks connecting two milestones (with the predecessor being a prerequisite for the successor), and with arc weights corresponding to task completion times (see Figure 2.3). Note that the project is feasible if and only if its PERT

Task	Prerequisites	Time (days)
A	(none)	3
B	(none)	2
C	A, B	2
D	C	4
E	C	3
F	D, E	2
G	C	3
H	G	1
I	F	5
J	H, I	2
K	J	10

Table 2.1: The tasks, prerequisite structure, and timings required to complete an example project.

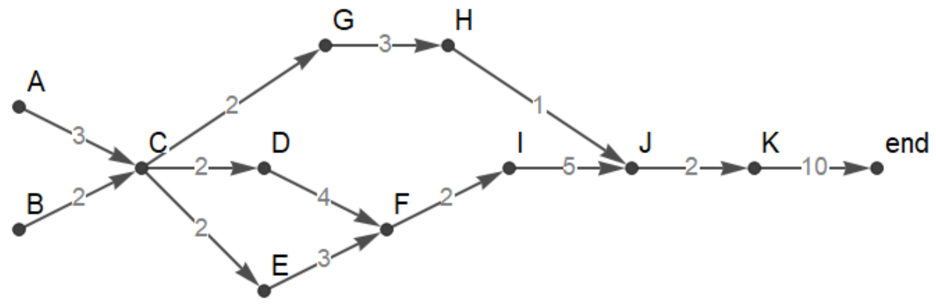

Figure 2.3: The PERT chart for the project defined in Table 2.1. Each node corresponds to the beginning of a task, plus an extra "end" node representing the completion of the entire project. Arc weights correspond to the time for the project at the tail of the arc.

chart is acyclic, since a directed cycle of dependencies would make it impossible to begin a portion of the project.

Some of the basic questions we may want to answer about this type of project include: How long will the project take to complete? Which tasks are the most "important" in terms of their effect on the overall project time? Assuming that tasks can be completed in parallel, both of these questions can be answered using **critical path analysis**.

A **critical path** is a **longest path** in the PERT chart (see Figure 2.4). A task with multiple prerequisites can only begin after *all* prerequisites have been completed, so a milestone in the PERT chart can only be reached after *all* incoming paths have been traversed. As such, the earliest start time for any given task is equal to the *longest* path length from any of its predecessors. It follows that the total project time equals the total length of the longest path in the network, since this represents the longest string of consecutive tasks that cannot be completed in parallel. It further follows that minor changes to task timings only directly affect the total project time for tasks on a critical path.

Longest paths (and therefore critical paths) can be found efficiently in DAGs, for example by applying the following algorithm.

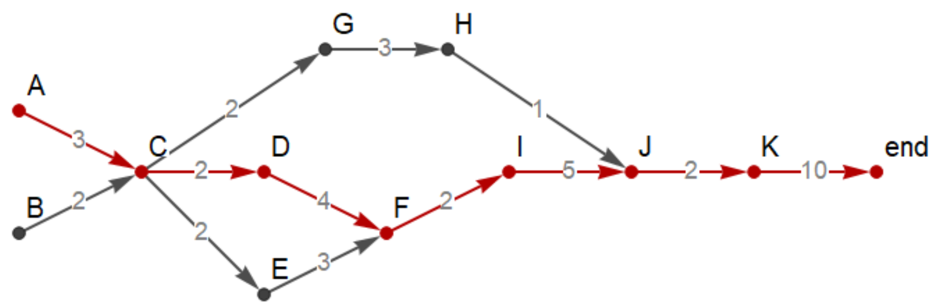

Figure 2.4: A PERT chart with a highlighted critical path. This represents the longest path in the network, and its total length is the total project completion time.

Longest Path Algorithm

```
1. Arrange all nodes in $V$ in topological order $(v_1, v_2, \ldots, v_n)$.
2. For all $v \in V$, set $L(v) = 0$ and $\text{pred}(v) = \text{null}$.
3. For each node $v_i$ in topological order $i = 1, 2, \ldots, n$, do the following:
    (a) For each node $v_j \in N^-(v_i)$, do the following:
        i. Test whether $L(v_j) + w(v_j, v_i) > L(v_i)$.
           If so, update $L(v_i)$ to $L(v_j) + w(v_j, v_i)$ and $\text{pred}(v_i)$ to $v_j$.
           Otherwise, leave $L(v_i)$ and $\text{pred}(v_i)$ alone.
4. Return the path lengths $L(v)$ and predecessors $\text{pred}(v)$ for all $v \in V$.
```

In short, this algorithm consists of searching the nodes in topological order. Each node v begins with a label $L(v) = 0$ representing the tentative length of the longest path ending at v. When a node is searched, we consider all its predecessors and update its label to be the maximum path length possible via any predecessor. On termination, $L(v)$ equals the length of the longest path ending at v for all $v \in V$, and in a PERT chart the label of the final node corresponds to the total project time (see Figure 2.5).

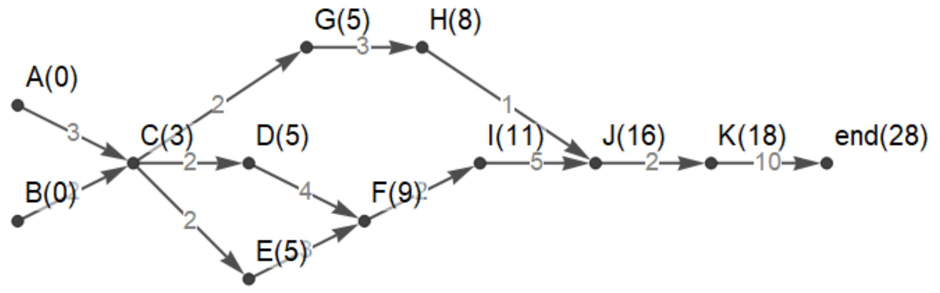

Figure 2.5: A PERT chart with earliest start times recorded next to each node. These times represent the lengths of the longest paths ending at each node, each of which is dictated by the maximum time via any predecessor task.

3 Strong Connectivity

The following notes are drawn primarily from *Introduction to Graph Theory*, by Douglas West [8], with applications and implementation details drawn from *Graph Theory and its Applications to Problems of Society*, by Fred Roberts [7].

We've previously defined what it means for a graph to be **connected**, namely that there exist a path between every pair of vertices. In a digraph, we also need to take *directions* of paths into account. More precisely, a digraph $G = (V, E)$ is called **strongly connected** if there is a *directed* path from every node $u \in V$ to every other node $v \in V$. It is **weakly connected** if its underlying graph is connected (see Figure 3.1).

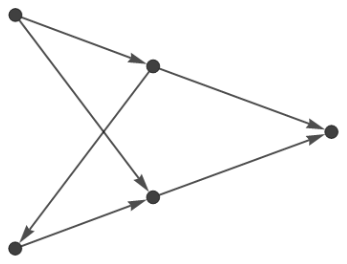

Figure 3.1: A digraph that is weakly connected but not strongly connected.

A **strongly connected orientation** (a.k.a. **strong orientation**) of an undirected graph G is an orientation of its edges such that the resulting digraph is stronglyc onnected. The topic of strong orientations was originally developed to study the *one-way street problem*, which asks whether it is possible for a given road network to be given a one-way assignment for every street such that every location can still be reached from every other location. For some graphs a strong orientation is possible, but for others it is not (see Figure 3.2).

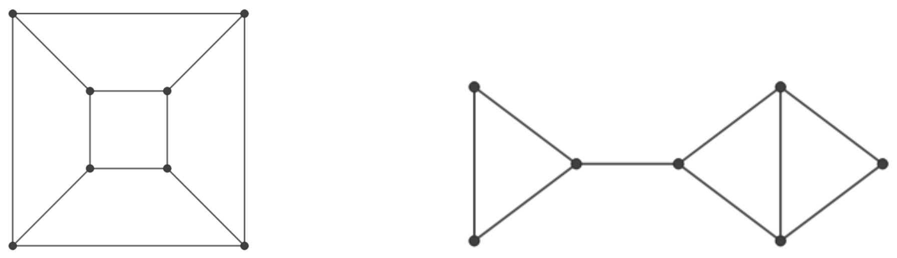

Figure 3.2: The graph on the left has a strong orientation, e.g. by orienting the inner and outer squares as directed cycles and alternating the diagonal edges between pointing towards or away from the inner cycle. The graph on the right cannot have a strong orientation due to the cut-edge bridging two components, since either choice of orientation leaves one part of the graph unreachable from the other.

It can immediately be seen that there are two necessary conditions for G to have a strong orientation: G must be connected (weak connectivity is required for strong connectivity) and it cannot have any cut-edges (since either orientation for a cut-edge leaves one part of the graph inaccessible from the other). Surprisingly this necessary condition also turns out to be *sufficient*, and this characterization for strong orientations is given by the following.

Theorem 3.1 (Robbins' Theorem). A graph G has a strong orientation if and only if it is connected and has no cut-edges.

The forward implication is obvious from the above observations, but the reverse implication is challenging to show directly. In order to prove Robbins' Theorem we need to introduce the ideas of *ears* and *ear decompositions*.

An **ear** of G is a subgraph $P \subseteq G$ consisting of a path that is part of a cycle, whose internal vertices all have degree 2 in G, and which is maximal with respect to the previous properties. The path may be closed, in which case the ear may be a simple cycle. An **ear decomposition** of a graph G is a partition of G into edge-disjoint subgraphs $P_0, P_1, \dots, P_k$ such that P_0 is a simple cycle and, for all $i = 1, 2, \dots, k$, P_i is an ear of the subgraph $P_0 \cup P_1 \cup \dots \cup P_i$ (see Figure 3.3).

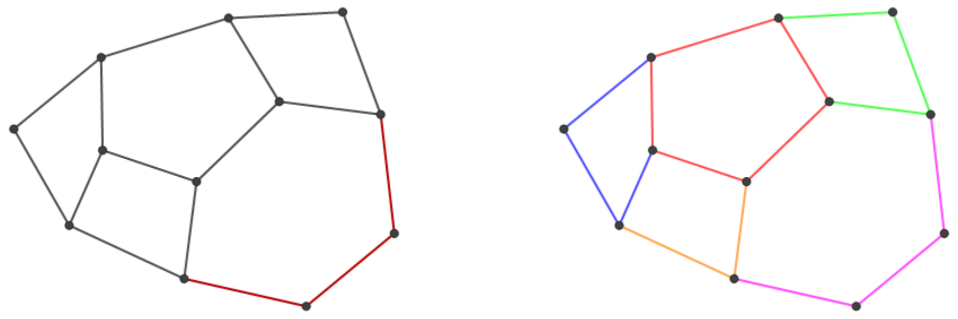

Figure 3.3: (Left) A graph with an ear highlighted. It is a path subgraph, it is part of a cycle, its internal vertices all have degree 2, and it is *maximal* in the sense that it cannot be extended while maintaining all of these properties. (Right) An ear decomposition of a graph. Colors represent the different ears in the decomposition, in the order: P_0, P_1, P_2, P_3, P_4.

Ears and ear decompositions were originally defined in Herbert Robbins' 1939 paper studying the one-way street problem [6], though they have found uses in other areas of discrete mathematics since then. To prove Robbins' Theorem we will actually show the stronger result that the following *three* statements about an undirected graph G are equivalent:

(a) G has a strong orientation.

(b) G is connected and has no cut-edges.

(c) G has an ear decomposition.

The proof will consist of three parts: showing that (a) implies (b), that (b) implies (c), and that (c) implies (a), which will prove that all three conditions are equivalent.

Lemma 3.2. If G has a strong orientation, then it is connected and has no cut-edges.

Proof. We will show the contrapositive of this statement: if G is disconnected or has a cut-edge, then it cannot have a strong orientation. First, suppose G is disconnected. Weak connectivity is required for strong connectivity, so G cannot have a strong orientation.

Next, suppose G has a cut-edge $\{u, v\} \in E(G)$. Orienting the arc from u to v produces a directed path from u to v, but there cannot be a path from v to u since this would require a path from v to u not involving the edge $\{u, v\}$. An edge is a cut-edge if and only if it is not part of any cycle, so there cannot be any alternate paths from v to u, and thus orienting $\{u, v\}$ from u to v does not yield a strong orientation. By symmetry orienting from v to u also does not yield a strong orientation, thus no strong orientation of G exists. □

Next, to prove that (c) implies (a) requires first establishing a proposition.

Proposition 3.3. Adding an ear to a graph with a strong orientation yields a larger graph with a strong orientation.[1]

Proof. [redacted] □

With this proposition in place, we can prove the necessary lemma.

Lemma 3.4. If G has an ear decomposition, then it has a strong orientation.[2]

Proof. [redacted] □

Finally, we can prove that (b) implies (c) to complete the implications.

Lemma 3.5. If G is connected and has no cut-edges, then it has an ear decomposition.

We will prove this algorithmically, by showing that the following procedure produces an ear decomposition (assuming G is connected and has no cut-edges). Throughout the following discussion, let $G_i = P_0 \cup P_1 \cup \cdots \cup P_i$ be the subgraph of G consisting of the first i ears (plus the initial cycle) in the ear decomposition.

Ear Decomposition Algorithm

1. Define $G_0 = P_0$ as any cycle in G.
2. Repeat the following for $i = 1, 2, \ldots$ until $G_i = G$:
   (a) Find an edge $\{u, v\} \in E(G) \setminus E(G_i)$ such that $u \in V(G_i)$.
   (b) $\{u, v\}$ must be part of a cycle C in G. Define P_{i+1} as the part of C traversed in following C starting at u until returning to G_i.
3. Return $P_0, P_1, \ldots, P_k$ as the components of an ear decomposition of G.

[1]Proof redacted due to homework.
[2]See footnote 1.

To prove Lemma 3.5, we will prove that this algorithm is well-defined, that it terminates in finite time, and that it produces an ear decomposition of G.

Proof. First, note that the initialization of the algorithm is well-defined. G being connected and lacking cut-edges implies that every edge $e \in E(G)$ is part of a cycle, so there must exist some cycle to choose as P_0.

Next, note that the main loop is well-defined. If the algorithm has not yet terminated, then there must exist some edges in $E(G) \setminus E(G_i)$, and since G is connected, at least one of these edges must have an endpoint in G_i, therefore it is always possible to choose an uncaptured edge $\{u, v\} \in E(G) \setminus E(G_i)$ such that $u \in V(G_i)$. Further, since $\{u, v\}$ is not a cut-edge, it is part of a cycle, and this cycle can be followed through the edges in $E(G) \setminus E(G_i)$ until returning to a vertex in $V(G_i)$.

Finally, each iteration of the main loop adds an ear with at least 1 edge to the decomposition, and so it must terminate after at most $|E(G)|$ iterations. The previous arguments show that, until the decomposition encompasses all of G, another iteration will still be possible, so it can only terminate when $G_i = G$. □

Lemmas 3.2, 3.4, and 3.5, together, prove Robbins' Theorem. While the initial statement of Theorem 3.1 answers the question of when it is possible to orient *all* edges in a graph to yield a strongly connected digraph, many practical instances of graph orientation problems only require *some* edges in the graph to be oriented. For example, given a graph G, we might want to know whether it is possible to orient *just one* edge while still leaving a directed path between every pair of vertices. This could be useful, for example, to determine whether one lane of a street can be closed for repairs. It's not to hard to see that the characterization for this property is identical to the characterization for strong orientations.

Observation 3.6. *All* edges in G can be oriented to yield a strongly connected digraph if and only if *any* edge in G can be oriented to yield a strongly connected mixed graph.

This is because, as argued earlier, any orientation chosen for a cut-edge leaves part of the graph unreachable from another part. Therefore if it is possible to orient any one edge in G to yield a strongly connected mixed graph, G cannot contain any cut-edges, which is exactly the characterization of G having a strong orientation.

Finally, we note an algorithm for producing strong orientations (when they exist) based on *depth-first search (DFS).*

Strong Orientation Algorithm

1. Perform a DFS of G starting at any arbitrary node, and label the nodes $1, 2, \ldots, n$ in order of traversal.

2. For each edge in G that is part of the DFS tree, orient the edge from lower label to higher label.

3. For each edge in G that is not part of the DFS tree, orient the edge from higher label to lower label.

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, Upper Saddle River, NJ, 1993. ISBN 0-13-617549-X.

[2] J. A. Dossey, A. D. Otto, L. E. Spence, and C. Vanden Eynden. *Discrete Mathematics*. Pearson, 5th edition, 2006. ISBN 0-321-30515-9.

[3] R. Johnsonbaugh. *Discrete Mathematics*. Pearson, 8th edition, 2023. ISBN 9780137848577. URL `https://www.pearson.com/en-us/subject-catalog/p/discrete-mathematics/P200000006219`.

[4] A. B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962. `doi: 10.1145/368996.369025`.

[5] J. Morris. *Combinatorics: an upper-level introductory course in enumeration, graph theory, and design theory*. version 2.1.1, 2023. URL `https://www.cs.uleth.ca/~morris/Combinatorics/html/frontmatter-1.html`.

[6] H. E. Robbins. A theorem on graphs, with an application to a problem of traffic control. *The American Mathematical Monthly*, 46(5):281–283, 1939. `doi: 10.2307/2303897`.

[7] F. S. Roberts. *Graph Theory and Its Applications to Problems of Society*. SIAM, Philadelphia, PA, 1978. ISBN 0-89871-026-X.

[8] D. B. West. *Introduction to Graph Theory*. Prentice Hall, Upper Saddle River, NJ, 2nd edition, 2001. ISBN 0-13-014400-2.