Chapter 3

Graphs, Part I: Basic Notions

3.1 Why Graphs? Some Motivations

Graphs are mathematical structures that have many applications to computer science, electrical engineering and more widely to engineering as a whole, but also to sciences such as biology, linguistics, and sociology, among others.

For example, relations among objects can usually be encoded by graphs.

Whenever a system has a notion of state and a state transition function, graph methods may be applicable.

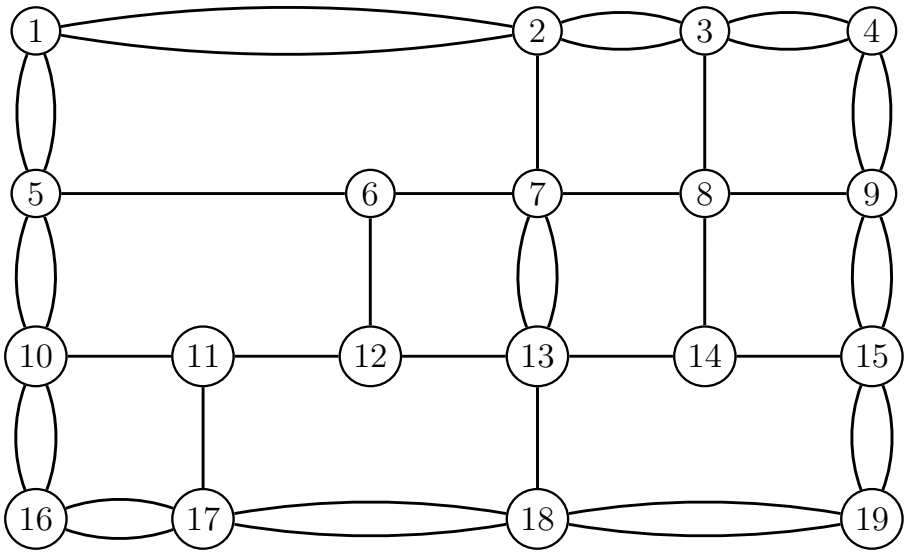

Figure 3.1: An undirected graph modeling a city map

Certain problems are naturally modeled by undirected graphs whereas others require directed graphs. Let us give a concrete example.

Suppose a city decides to create a public-transportation system.

The *undirected graph* of Figure 3.1 represents a map of some busy streets in a city.

The city decides to improve the traffic by making these streets *one-way* streets.

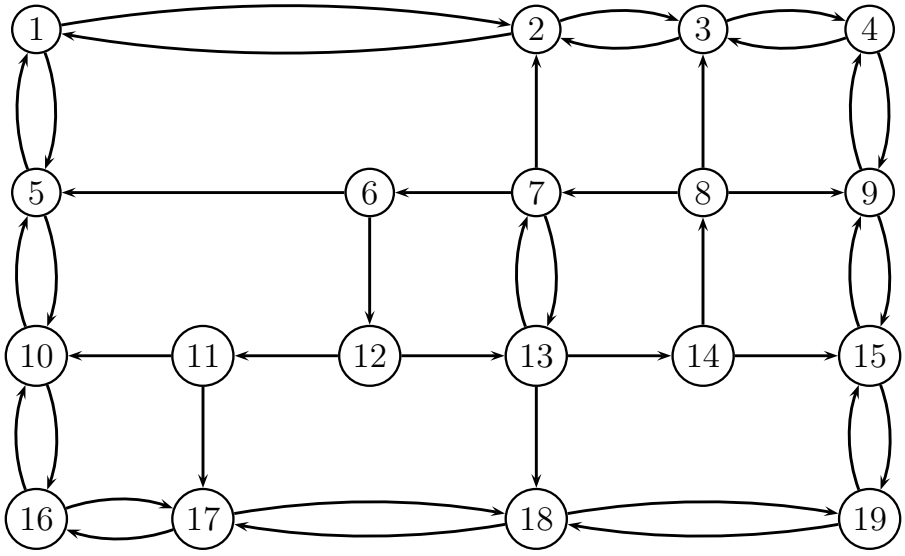

Figure 3.2: A choice of one-way streets

The problem requires finding a directed graph, given an undirected graph.

However, a good choice of orientation should allow one to travel between any two locations. We say that the resulting *directed graph* is *strongly connected*.

A possibility of orienting the streets is shown in Figure 3.2.

Did the engineers do a good job or are there locations such that it is impossible to travel from one to the other while respecting the one-way signs?


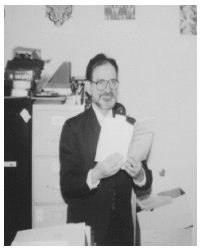

Figure 3.3: Claude Berge, 1926-2002 (left) and Frank Harary, 1921-2005 (right)

There is a peculiar aspect of graph theory having to do with its terminology.

Indeed, unlike most branches of mathematics, it appears that the terminology of graph theory is not standardized, yet.

This can be quite confusing to the beginner who has to struggle with many different and often inconsistent terms denoting the same concept, one of the worse being the notion of a *path*.

Our attitude has been to use terms that we feel are as simple as possible.

Many books begin by discussing undirected graphs and introduce directed graph only later on.

We disagree with this approach.

Indeed, we feel that the notion of a directed graph is more fundamental than the notion of an undirected graph.

For one thing, a unique undirected graph is obtained from a directed graph by forgetting the direction of the arcs, whereas there are many ways of orienting an undirected graph.

Also, in general, we believe that most definitions about directed graphs are cleaner than the corresponding ones for undirected graphs (for instance, we claim that the definition of a directed graph is simpler than the definition of an undirected graph, and similarly for paths).

3.2 Directed Graphs

Informally, a directed graph consists of a set of *nodes* together with a set of *oriented arcs* (also called *edges*) between these nodes.

Every arc has a single *source* (or initial point) and a single *target* (or endpoint), both of which are nodes.

There are various ways of formalizing what a directed graph is and some decisions must be made. Two issues must be confronted:

1. Do we allow "loops," that is, arcs whose source and target are identical?
2. Do we allow "parallel arcs," that is distinct arcs having the same source and target?

Since every binary relation on a set can be represented as a directed graph with loops, our definition allows loops.

Since the directed graphs used in automata theory must accomodate parallel arcs (usually labeled with different symbols), our definition also allows parallel arcs.

Before giving a formal definition, let us say that graphs are usually depicted by drawings (graphs!) where the nodes are represented by circles containing the node name and oriented line segments labeled with their arc name (see Figure 3.4).

Definition 3.2.1 A *directed graph* (or *digraph*) is a quadruple, $G = (V, E, s, t)$, where V is a set of *nodes or vertices*, E is a set of *arcs or edges* and $s, t \colon E \to V$ are two functions, s being called the *source function* and t the *target function*. Given an edge $e \in E$, we also call $s(e)$ the *origin* or *source* of e, and $t(e)$ the *endpoint* or *target* of e.

If the context makes it clear that we are dealing only with directed graphs, we usually say simply "graph" instead of "directed graph".

A directed graph, $G = (V, E, s, t)$, is *finite* iff both V and E are finite. In this case, $|V|$, the number of nodes of G is called the *order* of G.

Example: Let G_1 be the directed graph defined such that

$$E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9\},$$
$$V = \{v_1, v_2, v_3, v_4, v_5, v_6\}, \text{ and}$$
$$s(e_1) = v_1, s(e_2) = v_2, s(e_3) = v_3, s(e_4) = v_4,$$
$$s(e_5) = v_2, s(e_6) = v_5, s(e_7) = v_5, s(e_8) = v_5, s(e_9) = v_6$$
$$t(e_1) = v_2, t(e_2) = v_3, t(e_3) = v_4, t(e_4) = v_2,$$
$$t(e_5) = v_5, t(e_6) = v_5, t(e_7) = v_6, t(e_8) = v_6, t(e_9) = v_4.$$

The graph G_1 is represented by the diagram shown in Figure 3.4.

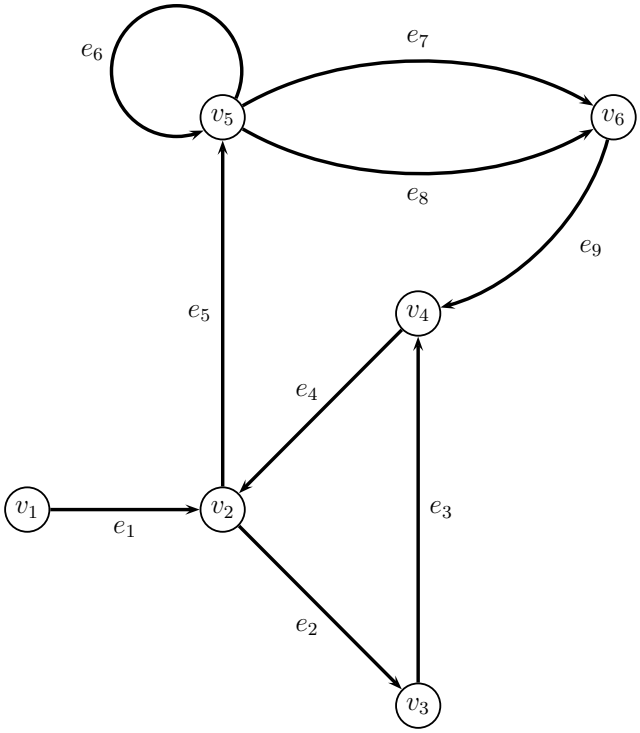

Figure 3.4: A directed graph, G_1

It should be noted that there are many different ways of "drawing" a graph.

Obviously, we would like as much as possible to avoid having too many intersecting arrows but this is not always possible if we insist in drawing a graph on a sheet of paper (on the plane).

Definition 3.2.2 Given a directed graph, G, an edge, $e \in E$, such that $s(e) = t(e)$ is called a *loop* (or *self-loop*). Two edges, $e, e' \in E$ are said to be *parallel edges* iff $s(e) = s(e')$ and $t(e) = t(e')$. A directed graph is *simple* iff it has no parallel edges.

Remarks:

1. The functions s, t need not be injective or surjective. Thus, we allow "isolated vertices", that is, vertices that are not the source or the target of any edge.

2. When G is simple, every edge, $e \in E$, is uniquely determined by the ordered pair of vertices, (u, v), such that $u = s(e)$ and $v = t(e)$.

   In this case, we may denote the edge e by (uv) (some books also use the notation uv).

   Also, a graph without parallel edges can be defined as a pair, (V, E), with $E \subseteq V \times V$. In other words, a simple graph is equivalent to a binary relation on a set ($E \subseteq V \times V$). This definition is often the one used to define directed graphs.

3. Given any edge, $e \in E$, the nodes $s(e)$ and $t(e)$ are often called the *boundaries* of e and the expression $t(e) - s(e)$ is called the *boundary of e*.

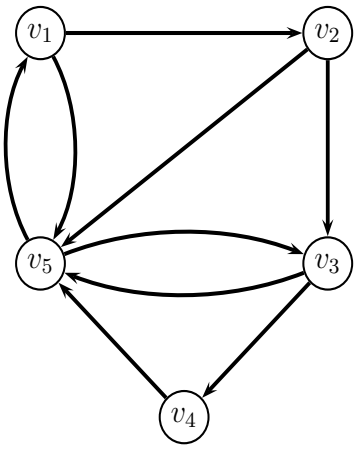

Figure 3.5: A directed graph, G_2

4. Given a graph, $G = (V, E, s, t)$, we may also write $V(G)$ for V and $E(G)$ for E. Sometimes, we even drop s and t and write simply $G = (V, E)$ instead of $G = (V, E, s, t)$.

5. Some authors define a simple graph to be a graph without loops and without parallel edges.

Observe that the graph G_1 has the loop e_6 and the two parallel edges e_7 and e_8.

When we draw pictures of graphs, we often omit the edge names (sometimes even the node names) as illustrated in Figure 3.5.

Definition 3.2.3 Given a directed graph, G, for any edge $e \in E$, if $u = s(e)$ and $v = t(e)$, we say that

(i) The nodes u and v are *adjacent*

(ii) The nodes u and v are *incident to the arc e*

(iii) The arc e is *incident to the nodes u and v*

(iv) Two edges, $e, e' \in E$ are *adjacent* if they are incident to some common node (that is, either $s(e) = s(e')$ or $t(e) = t(e')$ or $t(e) = s(e')$ or $s(e) = t(e')$).

For any node, $u \in V$, set

(a) $d_G^+(u) = |\{e \in E \mid s(e) = u\}|$, the *outer half-degree or outdegree of u*

(b) $d_G^-(u) = |\{e \in E \mid t(e) = u\}|$, the *inner half-degree or indegree of u*

(c) $d_G(u) = d_G^+(u) + d_G^-(u)$, the *degree of u*.

A graph is *regular* iff every node has the same degree.

Note that d^+_G (respectively $d^-_G(u)$) counts the number of arcs "coming out from u", that is, whose source is u (resp. counts the number of arcs "coming into u", that is, whose target is u). For example, in the graph of Figure 3.5, $d^+_{G_2}(v_1) = 2$, $d^-_{G_2}(v_1) = 1$, $d^+_{G_2}(v_5) = 2$, $d^-_{G_2}(v_5) = 4$, $d^+_{G_2}(v_3) = 2$, $d^-_{G_2}(v_3) = 2$. Neither G_1 nor G_2 are regular graphs.

The first result of graph theory is the following simple but very useful proposition:

Proposition 3.2.4 *For any finite graph, $G = (V, E, s, t)$, we have*

$$\sum_{u \in V} d^+_G(u) = \sum_{u \in V} d^-_G(u).$$

Corollary 3.2.5 *For any finite graph, $G = (V, E, s, t)$, we have*

$$\sum_{u \in V} d_G(u) = 2|E|,$$

that is, the sum of the degrees of all the nodes is equal to twice the number of edges.

Corollary 3.2.6 *For any finite graph, $G = (V, E, s, t)$, there is an even number of nodes with an odd degree.*

The notion of homomorphism and isomorphism of graphs is fundamental.

Definition 3.2.7 Given two directed graphs, $G_1 = (V_1, E_1, s_1, t_1)$ and $G_2 = (V_2, E_2, s_2, t_2)$, a *homomorphism* (or *morphism*), $f : G_1 \to G_2$, *from* G_1 *to* G_2 is a pair, $f = (f^v, f^e)$, with $f^v : V_1 \to V_2$ and $f^e : E_1 \to E_2$ preserving incidence, that is, for every edge, $e \in E_1$, we have

$$s_2(f^e(e)) = f^v(s_1(e)) \quad \text{and} \quad t_2(f^e(e)) = f^v(t_1(e)).$$

These conditions can also be expressed by saying that the following two diagrams commute:

$$\begin{array}{ccc} E_1 & \xrightarrow{f^e} & E_2 \\ {\scriptstyle s_1}\downarrow & & \downarrow{\scriptstyle s_2} \\ V_1 & \xrightarrow[f^v]{} & V_2 \end{array} \qquad \begin{array}{ccc} E_1 & \xrightarrow{f^e} & E_2 \\ {\scriptstyle t_1}\downarrow & & \downarrow{\scriptstyle t_2} \\ V_1 & \xrightarrow[f^v]{} & V_2. \end{array}$$

Given three graphs, G_1, G_2, G_3 and two homomorphisms, $f\colon G_1 \to G_2$ and $g\colon G_2 \to G_3$, with $f = (f^v, f^e)$ and $g = (g^v, g^e)$, it is easily checked that $(g^v \circ f^v, g^e \circ f^e)$ is a homomorphism from G_1 to G_3.

The homomorphism $(g^v \circ f^v, g^e \circ f^e)$ is denoted $g \circ f$.

Also, for any graph, G, the map $\mathrm{id}_G = (\mathrm{id}_V, \mathrm{id}_E)$ is a homomorphism called the *identity homomorphism*. Then, a homomorphism, $f\colon G_1 \to G_2$, is an *isomorphism* iff there is a homomorphism, $g\colon G_2 \to G_1$, such that

$$g \circ f = \mathrm{id}_{G_1} \quad \text{and} \quad f \circ g = \mathrm{id}_{G_2}.$$

In this case, g is unique and it is called the *inverse* of f and denoted f^{-1}. If $f = (f^v, f^e)$ is an isomorphism, we see immediately that f^v and f^e are bijections.

Checking whether two finite graphs are isomorphic is not as easy as it looks.

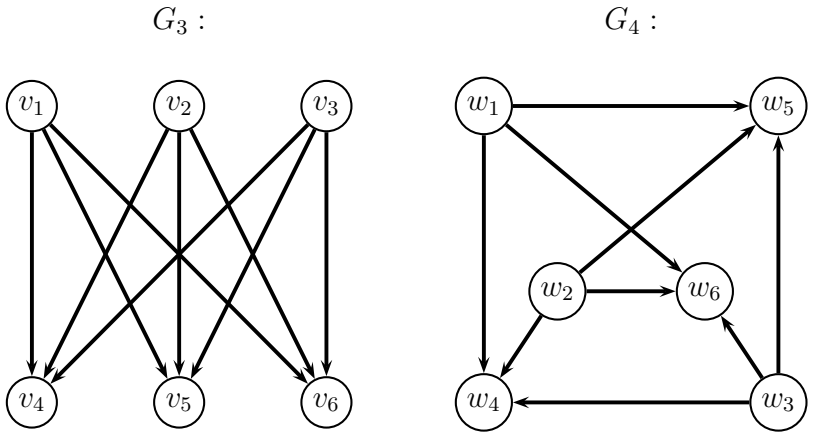

Figure 3.6: Two isomorphic graphs, G_3 and G_4

In fact, *no general efficient algorithm for checking graph isomorphism* is known at this time and determining the exact complexity of this problem is a *major open question in computer science*.

For example, the graphs G_3 and G_4 shown in Figure 3.6 are isomorphic.

The bijection f^v is given by $f^v(v_i) = w_i$, for $i = 1, \ldots, 6$ and the reader will easily figure out the bijection on arcs. As we can see, isomorphic graphs can look quite different.

3.3 Paths in Digraphs; Strongly Connected Components

Many problems about graphs can be formulated as path existence problems.

Given a directed graph, G, intuitively, a path from a node u to a node v is a way to travel from u in v by following edges of the graph that "link up correctly".

Unfortunately, if we look up the definition of a path in two different graph theory books, we are almost guaranteed to find different and usually clashing definitions!

The terminology that we have chosen may not be standard, but it is used by a number of authors (some very distinguished, for example, Fields medalists!) and we believe that it is less taxing on one's memory (however, this point is probably the most debatable).

Definition 3.3.1 Given any digraph, $G = (V, E, s, t)$, and any two nodes, $u, v \in V$, a *path from u to v* is a triple, $\pi = (u, e_1 \cdots e_n, v)$, where $n \geq 1$ and $e_1 \cdots e_n$ is a sequence of edges, $e_i \in E$ (*i.e.*, a nonempty string in E^*), such that

$$s(e_1) = u;\ t(e_n) = v;\ t(e_i) = s(e_{i+1}),\ 1 \leq i \leq n-1.$$

We call n the *length of the path π* and we write $|\pi| = n$. When $n = 0$, we have the *null path*, (u, ϵ, u), from u to u (recall, ϵ denotes the empty string); the null path has length 0. If $u = v$, then π is called a *closed path*, else an *open path*.

The path, $\pi = (u, e_1 \cdots e_n, v)$, determines the sequence of nodes, $\text{nodes}(\pi) = \langle u_0, \ldots, u_n \rangle$, where $u_0 = u$, $u_n = v$ and $u_i = t(e_i)$, for $1 \leq i \leq n$. We also set $\text{nodes}((u, \epsilon, u)) = \langle u, u \rangle$.

A path, $\pi = (u, e_1 \cdots e_n, v)$, is *edge-simple*, for short, *e-simple* iff $e_i \neq e_j$ for all $i \neq j$ (*i.e.*, no edge in the path is used twice).

A path, π, from u to v is *simple* iff no vertex in $\text{nodes}(\pi)$ occurs twice, except possibly for u if π is closed.

Equivalently, if $\text{nodes}(\pi) = \langle u_0, \ldots, u_n \rangle$, then π is simple iff either

1. $u_i \neq u_j$ for all i, j with $i \neq j$ and $0 \leq i, j \leq n$, or π is closed *i.e*, $u_0 = u_n$, in which case
2. $u_i \neq u_0 \, (= u_n)$ for all i with $1 \leq i \leq n-1$, and $u_i \neq u_j$ for all i, j with $i \neq j$ and $1 \leq i, j \leq n-1$.

The null path, (u, ϵ, u), is considered e-simple and simple.

Remarks:

1. Other authors (such as Harary [10]) use the term *walk* for what we call a path. These authors also use the term *trail* for what we call an e-simple path and the term *path* for what we call a simple path. We decided to adopt the term "simple path" because it is prevalent in the computer science literature. However, note that Berge [2] and Sakarovitch [15] use the locution *elementary path* instead of simple path.
2. If a path, π, from u to v is simple, then every every node in the path occurs once except possibly u if $u = v$ so, every edge in π occurs exactly once. Therefore, *every simple path is an e-simple path*.

3. If a digraph is not simple, then even if a sequence of nodes is of the form nodes(π) for some path, that sequence of nodes does not uniquely determine a path.

   For example, in the graph of Figure 3.7, the sequence $\langle v_2, v_5, v_6 \rangle$ corresponds to the two distinct paths (v_2, e_5e_7, v_6) and (v_2, e_5e_8, v_6).

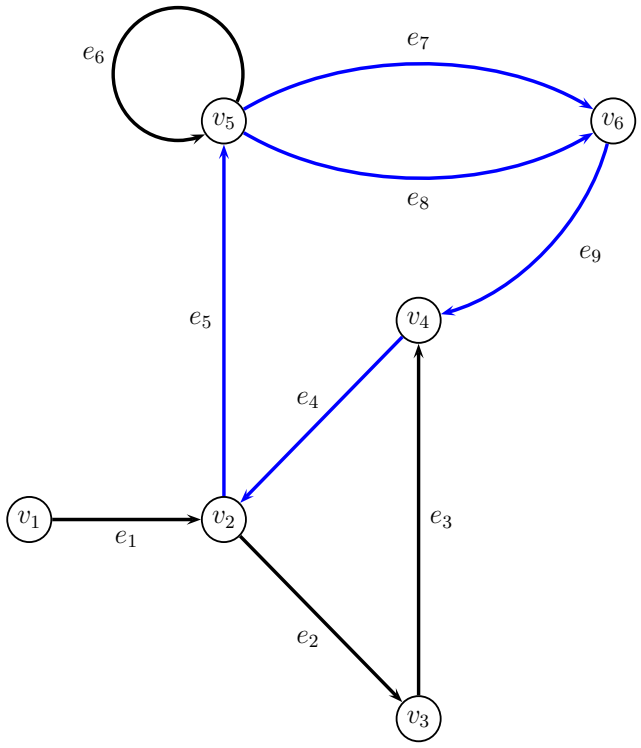

Figure 3.7: A path in a directed graph, G_1

In the graph G_1 from Figure 3.7,

$$(v_2, e_5e_7e_9e_4e_5e_8, v_6)$$

is a path from v_2 to v_6 which is neither e-simple nor simple.

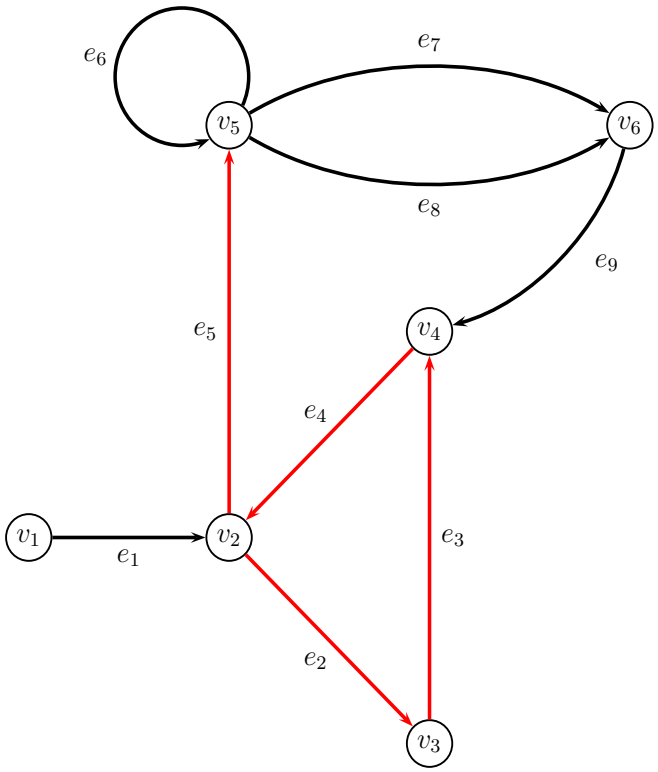

Figure 3.8: An e-simple path in a directed graph, G_1

The path

$$(v_2, e_2e_3e_4e_5, v_5)$$

is an e-simple path from v_2 to v_5 which is not simple and

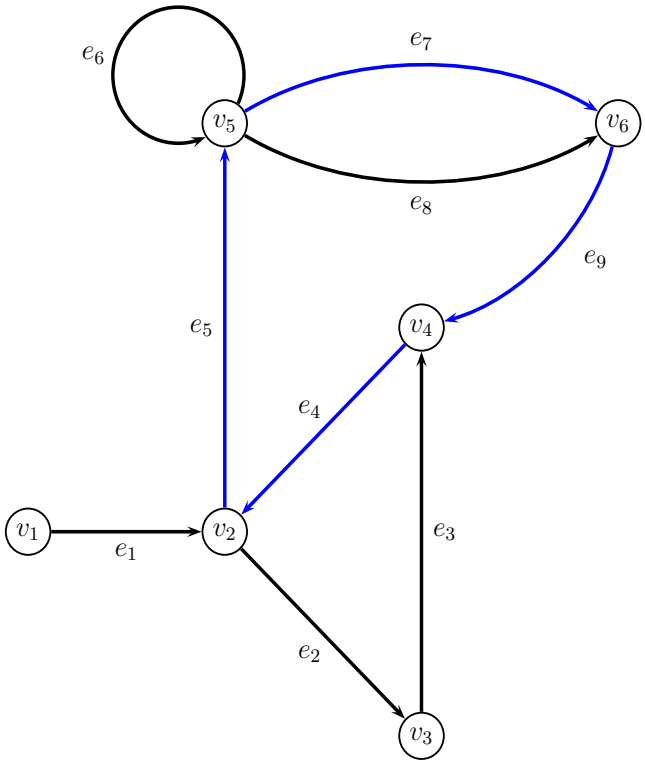

Figure 3.9: Simple paths in a directed graph, G_1

$$(v_2, e_5e_7e_9, v_4), \qquad (v_2, e_5e_7e_9e_4, v_2)$$

are simple paths, the first one open and the second one closed.

Recall the notion of subsequence of a sequence defined just before stating Theorem 2.9.17.

Then, if $\pi = (u, e_1 \cdots e_n, v)$ is any path from u to v in a digraph, G, a *subpath* of π is any path $\pi' = (u, e'_1 \cdots e'_m, v)$ such that $e'_1, \ldots, e'_m$ is a subsequence of $e_1, \ldots, e_n$. The following simple proposition is actually very important:

Proposition 3.3.2 *Let G be any digraph. (a) For any two nodes, u, v, in G, every non-null path, π, from u to v contains a simple non-null subpath.*

(b) If $|V| = n$, then every open simple path has length at most $n-1$ and every closed simple path has length at most n.

Like strings, paths can be concatenated.

Definition 3.3.3 Two paths, $\pi = (u, e_1 \cdots e_m, v)$ and $\pi' = (u', e'_1 \cdots e'_n, v')$ in a digraph G can be *concatenated* iff $v = u'$ in which case their *concatenation*, $\pi\pi'$, is the path

$$\pi\pi' = (u, e_1 \cdots e_m e'_1 \cdots e'_n, v').$$

We also let

$$(u, \epsilon, u)\pi = \pi = \pi(v, \epsilon, v).$$

Concatenation of paths is obviously associative and observe that $|\pi\pi'| = |\pi| + |\pi'|$.

Definition 3.3.4 Let $G = (V, E, s, t)$ be a digraph. We define the binary relation, $\widehat{C}_G$, on V as follows: For all $u, v \in V$,

$$u\widehat{C}_G v$$

iff there is a path from u to v and there is a path from v to u.

When $u\widehat{C}_G v$, we say that *u and v are strongly connected*.

The relation $\widehat{C}_G$ is what is called an equivalence relation. The notion of an equivalence relation is discussed extensively in Chapter 5 (Section 5.5) but because it is a very important concept, we explain briefly what it is right now.

Repeating Definition 5.5.1, a binary relation, R, on a set, X, is an *equivalence relation* iff it is *reflexive*, *transitive* and *symmetric*, that is:

(1) (*Reflexivity*): aRa, for all $a \in X$;

(2) (*Transitivity*): If aRb and bRc, then aRc, for all $a, b, c \in X$.

(3) (*symmetry*): If aRb, then bRa, for all $a, b \in X$.

The main property of equivalence relations is that they partition the set X into nonempty, pairwise disjoint subsets called equivalence classes:

For any $x \in X$, the set

$$[x]_R = \{y \in X \mid xRy\}$$

is the *equivalence class of* x.

Each equivalence class, $[x]_R$, is also denoted $\overline{x}_R$ and the subscript R is often omitted when no confusion arises.

For the reader's convenience, we repeat Proposition 5.5.3:

Let R be an equivalence relation on a set, X. For any two elements $x, y \in X$, we have

$$xRy \quad \textit{iff} \quad [x] = [y].$$

Moreover, the equivalences classes of R satisfy the following properties:

(1) $[x] \neq \emptyset$, for all $x \in X$;

(2) If $[x] \neq [y]$ then $[x] \cap [y] = \emptyset$;

(3) $X = \bigcup_{x \in X} [x]$.

The relation $\widehat{C}_G$ is reflexive because we have the null path from u to u, symmetric by definition, and transitive because paths can be concatenated.

The equivalence classes of the relation $\widehat{C}_G$ are called the *strongly connected components of G (SCC's)*.

A graph is *strongly connected* iff it has a single strongly connected component.

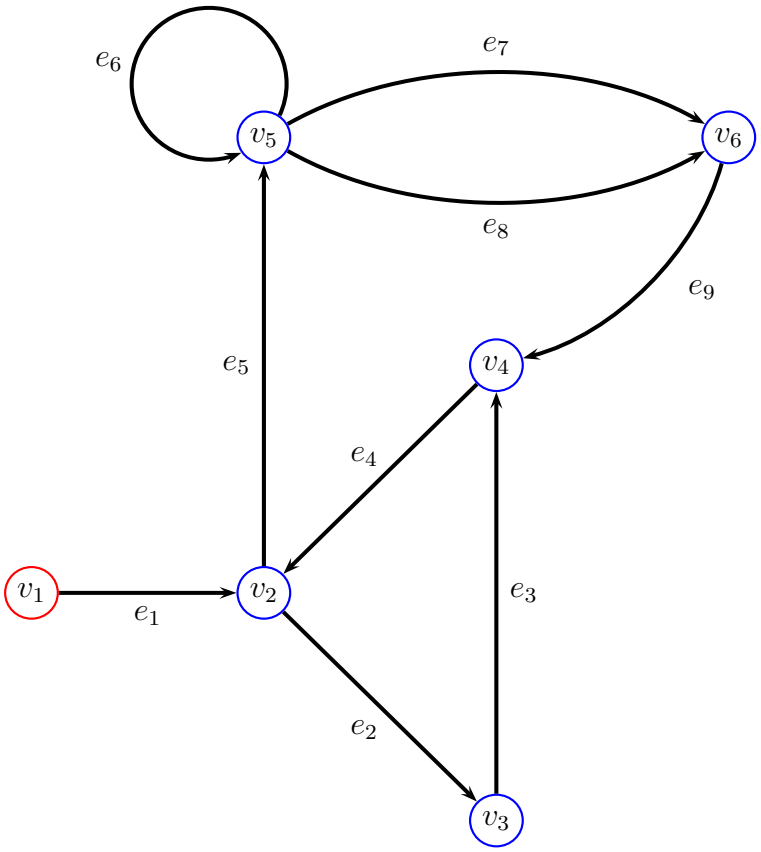

Figure 3.10: A directed graph, G_1, with two SCC's

For example, we see that the graph, G_1, of Figure 3.10 has two strongly connected components

$$\{v_1\}, \quad \{v_2, v_3, v_4, v_5, v_6\},$$

since there is a closed path

$$(v_4, e_4e_2e_3e_4e_5e_7e_9, v_4).$$

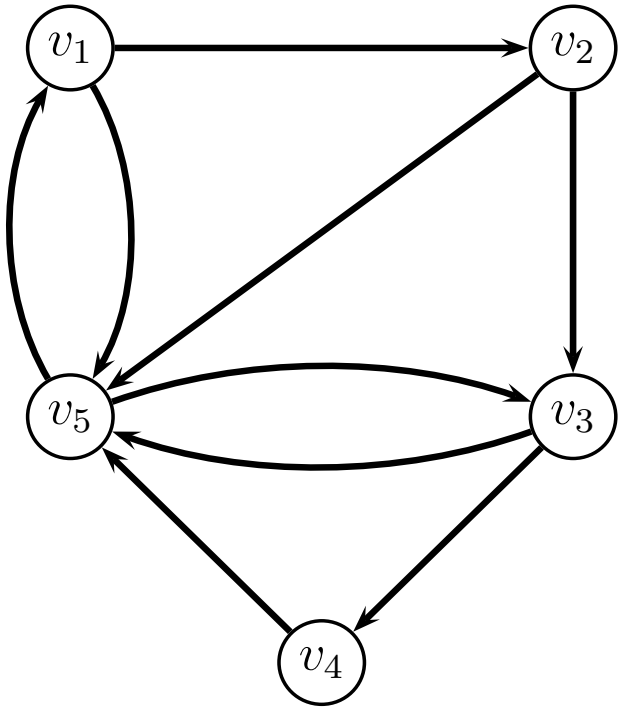

Figure 3.11: A strongly connected directed graph, G_2

The graph G_2 of Figure 3.11 is strongly connected.

Let us give a simple algorithm for computing the strongly connected components of a graph since this is often the key to solving many problems.

The algorithm works as follows: Given some vertex, $u \in V$, the algorithm computes the two sets, $X^+(u)$ and $X^-(u)$, where

$$X^+(u) = \{v \in V \mid \text{there exists a path from } u \text{ to } v\}$$
$$X^-(u) = \{v \in V \mid \text{there exists a path from } v \text{ to } u\}.$$

Then, it is clear that the connected component, $C(u)$, of u, is given by
$C(u) = X^+(u) \cap X^-(u)$.

For simplicity, we assume that $X^+(u), X^-(u)$ and $C(u)$ are represented by linear arrays.

In order to make sure that the algorithm makes progress, we used a simple marking scheme.

We use the variable *total* to count how many nodes are in $X^+(u)$ (or in $X^-(u)$) and the variable *marked* to keep track of how many nodes in $X^+(u)$ (or in $X^-(u)$) have been processed so far.

Whenever the algorithm considers some unprocessed node, the first thing it does is to increment *marked* by 1.

```
function strcomp(G: graph; u: node): set
  begin
    X⁺(u)[1] := u; X⁻(u)[1] := u; total := 1; marked := 0;
    while marked < total do
      marked := marked + 1; v := X⁺(u)[marked];
      for each e ∈ E
        if (s(e) = v) ∧ (t(e) ∉ X⁺(u)) then
          total := total + 1; X⁺(u)[total] := t(e) endif
      endfor
    endwhile;
    total := 1; marked := 0;
    while marked < total do
      marked := marked + 1; v := X⁻(u)[marked];
      for each e ∈ E
        if (t(e) = v) ∧ (s(e) ∉ X⁻(u)) then
          total := total + 1; X⁻(u)[total] := s(e) endif
      endfor
    endwhile;
    C(u) = X⁺(u) ∩ X⁻(u); strcomp := C(u)
  end
```

If we want to obtain all the strongly connected components (SCC's) of a finite graph, G, we proceed as follows:

Set $V_1 = V$, pick any node, v_1, in V_1 and use the above algorithm to compute the strongly connected component, C_1, of v_1.

If $V_1 = C_1$, stop. Otherwise, let $V_2 = V_1 - C_1$.

Again, pick any node, v_2 in V_2 and determine the strongly connected component, C_2, of v_2.

If $V_2 = C_2$, stop. Otherwise, let $V_3 = V_2 - C_2$, pick v_3 in V_3, and continue in the same manner as before.

Ultimately, this process will stop and produce all the strongly connected components $C_1, \ldots, C_k$ of G.

It should be noted that the function *strcomp* and the simple algorithm that we just described are "naive" algorithms that are not particularly efficient.

Their main advantage is their simplicity. There are more efficient algorithms, in particular, there is a beautiful algorithm for computing the SCC's due to Robert Tarjan.

Going back to our city traffic problem from Section 3.1, if we compute the strongly connected components for the proposed solution shown in Figure 3.2, we find the three SCC's shown in Figure 3.12.

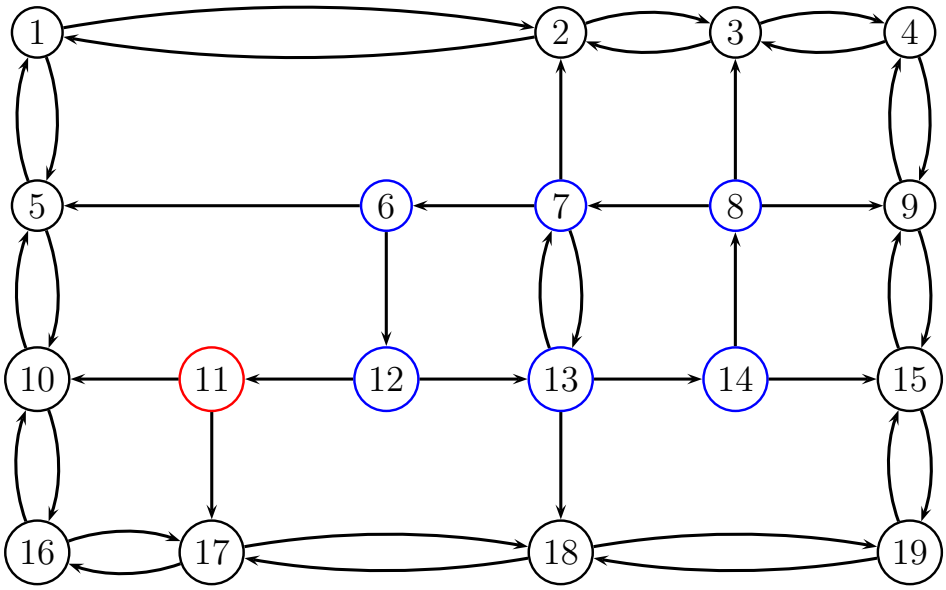

Figure 3.12: The strongly connected components of the graph in Figure 3.2

The three SCC's are

$$\{6, 7, 8, 12, 13, 14\}, \quad \{11\},$$

and

$$\{1, 2, 3, 4, 5, 9, 10, 15, 16, 17, 18, 19\}.$$

Therefore, the city engineers did not do a good job! We will show after proving Proposition 3.3.8 how to "fix" this faulty solution.

Closed e-simple paths also play an important role.

Definition 3.3.5 Let $G = (V, E, s, t)$ be a digraph. A *circuit* is a closed e-simple path (*i.e.*, no edge occurs twice) and a *simple circuit* is a simple closed path. The null path, (u, ϵ, u), is a simple circuit.

Remark: A closed path is sometimes called a *pseudo-circuit*. In a pseudo-circuit, some edge may occur more than once.

The significance of simple circuits is revealed by the next proposition.

Proposition 3.3.6 *Let G be any digraph. (a) Every circuit, π, in G is the concatenation of pairwise edge-disjoint simple circuits.*

(b) A circuit is simple iff it is a minimal circuit, that is, iff it does not contain any proper circuit.

Remarks:

1. If u and v are two nodes that belong to a circuit, π, in G, (*i.e.*, both u and v are incident to some edge in π), then u and v are strongly connected.

   Indeed, u and v are connected by a portion of the circuit π, and v and u are connected by the complementary portion of the circuit.

2. If π is a pseudo-circuit, the above proof shows that it is still possible to decompose π into simple circuits, but it may not be possible to write π as the concatenation of pairwise edge-disjoint simple circuits.

Given a graph, G, we can form a new and simpler graph from G by connecting the strongly connected components of G as shown below.

Definition 3.3.7 Let $G = (V, E, s, t)$ be a digraph. The *reduced graph*, $\widehat{G}$, is the simple digraph whose set of nodes, $\widehat{V} = V/\widehat{C}_G$, is the set of strongly connected components of V and whose set of edges, $\widehat{E}$, is defined as follows:

$$(\widehat{u}, \widehat{v}) \in \widehat{E} \quad \text{iff} \quad (\exists e \in E)(s(e) \in \widehat{u} \quad \text{and} \quad t(e) \in \widehat{v}),$$

where we denote the strongly connected component of u by $\widehat{u}$.

That $\widehat{G}$ is "simpler" than G is the object of the next proposition.

Proposition 3.3.8 *Let G be any digraph. The reduced graph, $\widehat{G}$, contains no circuits.*

Remark: Digraphs without circuits are called *DAG's*. Such graphs have many nice properties.

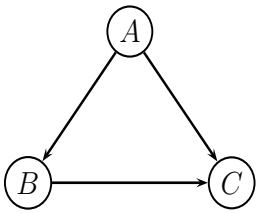

Figure 3.13: The reduced graph of the graph in Figure 3.12

In particular, it is easy to see that any finite DAG has nodes with no incoming edges. Then, it is easy to see that finite DAG's are basically collections of trees with shared nodes.

The reduced graph of the graph shown in Figure 3.12 is showed in Figure 3.13, where its SCC's are labeled A, B and C as shown below:

$$A = \{6, 7, 8, 12, 13, 14\}, \quad B = \{11\},$$

and

$$C = \{1, 2, 3, 4, 5, 9, 10, 15, 16, 17, 18, 19\}.$$

The locations in the component A are inaccessible.

Observe that changing the direction of *any* street between the strongly connected components A and C yields a solution, that is, a strongly connected graph. So, the engineers were not too far off after all!

A solution to our traffic problem obtained by changing the direction of the street between 13 and 18 is shown in Figure 3.14.

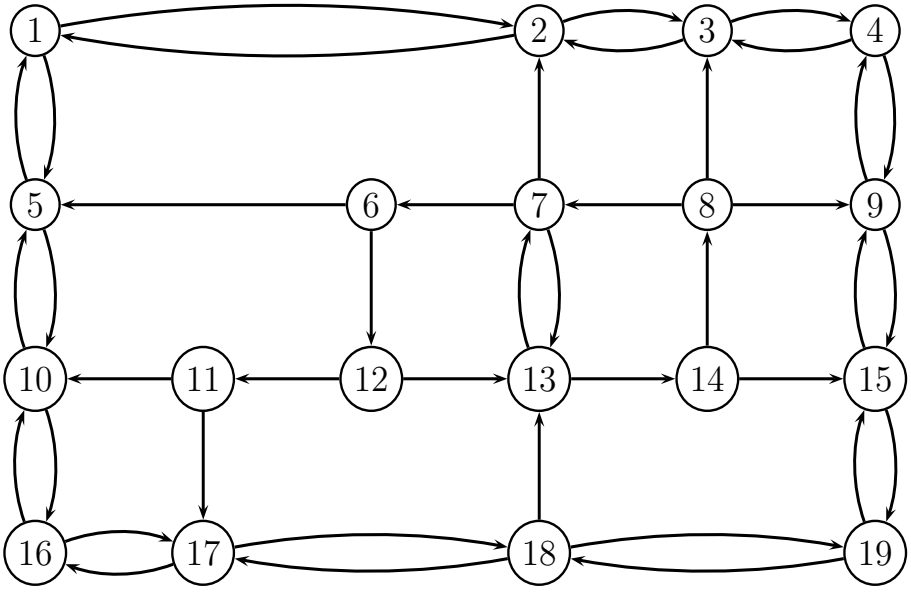

Figure 3.14: A good choice of one-way streets

Before discussing undirected graphs, let us collect various definitions having to do with the notion of subgraph.

Definition 3.3.9 Given any two digraphs, $G = (V, E, s, t)$ and $G' = (V', E', s', t')$, we say that *G' is a subgraph of G* iff $V' \subseteq V$, $E' \subseteq E$, s' is the retriction of s to E' and t' is the retriction of t to E'.

If G' is a subgraph of G and $V' = V$, we say that G' is a *spanning subgraph of G*.

Given any subset, V', of V, the *induced subgraph, $G\langle V'\rangle$, of G* is the graph $(V', E_{V'}, s', t')$ whose set of edges is

$$E_{V'} = \{e \in E \mid s(e) \in V'; t(e) \in V'\}.$$

(Clearly, s' and t' are the restrictions of s and t to $E_{V'}$, respectively.)

Given any subset, $E' \subseteq E$, the graph $G' = (V, E', s', t')$, where s' and t' are the restrictions of s and t to E', respectively, is called the *partial graph of G generated by E'*.

The graph, $(V', E' \cap E_{V'}, s', t')$, is a *partial subgraph of G* (here, s' and t' are the restrictions of s and t to $E' \cap E_{V'}$, respectively).

3.4 Undirected Graphs, Chains, Cycles, Connectivity

The edges of a graph express relationships among its nodes.

Sometimes, these relationships are not symmetric, in which case it is desirable to use directed arcs, as we have in the previous sections.

However, there is a class of problems where these relationships are naturally symmetric or where there is no a priori preferred orientation of the arcs.

For example, if V is the population of individuals that were students at Penn between 1900 until now and if we are interested in the relation where two people A and B are related iff they had the same professor in some course, then this relation is clearly symmetric.

As a consequence, if we want to find the set of individuals that are related to a given individual, A, it seems unnatural and, in fact, counter-productive, to model this relation using a directed graph.

As another example suppose we want to investigate the vulnerabilty of an internet network under two kinds of attacks:

(1) disabling a node; (2) cutting a link.

Again, whether of not a link between two sites is oriented is irrelevant.

What is important is that the two sites are either connected or disconnected.

These examples suggest that we should consider an "unoriented" version of a graph. How should we proceed?

One way to proceed is to still assume that we have a directed graph but to modify certain notions such as paths and circuits to account for the fact that such graphs are really "unoriented."

In particular, we should redefine paths to allow edges to be traversed in the "wrong direction".

Such an approach is possible but slightly awkward and ultimately it is really better to define undirected graphs.

However, to show that this approach is feasible, let us give a new definition of a path that corresponds to the notion of path in an undirected graph.

Definition 3.4.1 Given any digraph, $G = (V, E, s, t)$, and any two nodes, $u, v \in V$, a *chain* (or *walk*) *from u to v* is a sequence $\pi = (u_0, e_1, u_1, e_2, u_2, \ldots, u_{n-1}, e_n, u_n)$, where $n \geq 1$; $u_i \in V$; $e_j \in E$ and

$$u_0 = u;\ u_n = v \quad \text{and} \quad \{s(e_i), t(e_i)\} = \{u_{i-1}, u_i\},$$

$1 \leq i \leq n$. We call n the *length of the chain π* and we write $|\pi| = n$. When $n = 0$, we have the *null chain*, (u, ϵ, u), from u to u, a chain of length 0.

If $u = v$, then π is called a *closed chain*, else an *open chain*. The chain, π, determines the sequence of nodes, $\text{nodes}(\pi) = \langle u_0, \ldots, u_n \rangle$, with $\text{nodes}((u, \epsilon, u)) = \langle u, u \rangle$.

A chain, π, is *edge-simple*, for short, *e-simple* iff $e_i \neq e_j$ for all $i \neq j$ (*i.e.*, no edge in the chain is used twice).

A chain, π, from u to v is *simple* iff no vertex in $\text{nodes}(\pi)$ occurs twice, except possibly for u if π is closed. The null chain, (u, ϵ, u), is considered e-simple and simple.

The main difference between Definition 3.4.1 and Definition 3.3.1 is that Definition 3.4.1 *ignores the orientation*: in a chain, an edge may be traversed backwards, from its endpoint back to its source.

This implies that the reverse of a chain

$$\pi^R = (u_n, e_n, u_{n-1}, , \ldots, u_2, e_2, u_1, e_1, u_0)$$

is a chain from $v = u_n$ to $u = u_0$. In general, this fails for paths.

Note, as before, that if G is a simple graph, then a chain is more simply defined by a sequence of nodes

$$(u_0, u_1, \ldots, u_n).$$

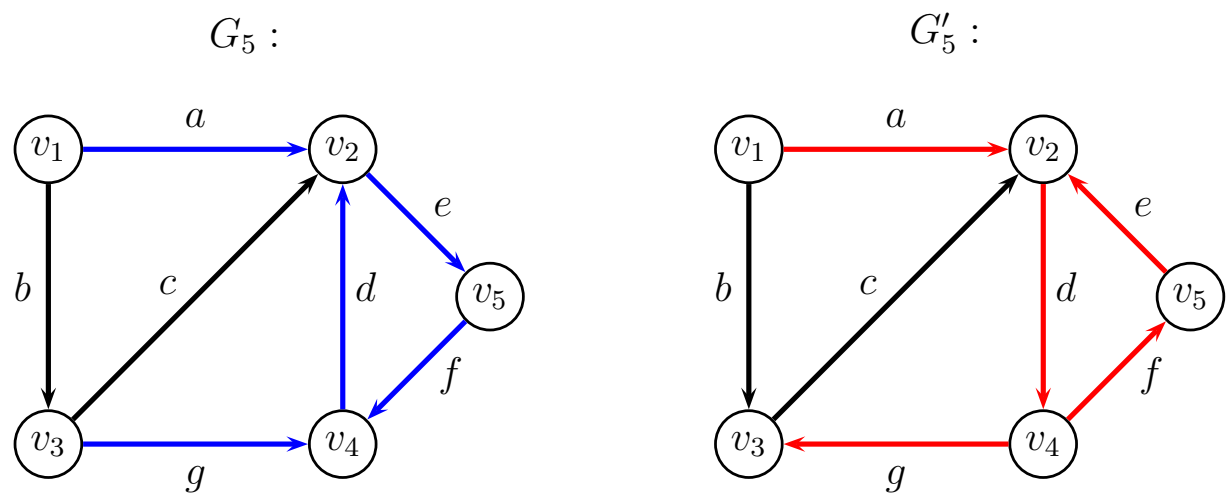

Figure 3.15: The Graphs G_5 and G'_5

For example, in the graph G_5 shown in Figure 3.15, we have the chains

$$(v_1, a, v_2, d, v_4, f, v_5, e, v_2, d, v_4, g, v_3),$$
$$(v_1, a, v_2, d, v_4, f, v_5, e, v_2),$$
$$(v_1, a, v_2, d, v_4, g, v_3)$$

from v_1 to v_3.

Note that none of these chains are paths. The graph G'_5 is obtained from the graph G_5 by reversing the direction of the edges, d, f, e, and g, so that the above chains are actually paths in G'_5.

The second chain is e-simple and the third is simple.

Chains are concatenated the same way as paths and the notion of subchain is analogous to the notion of subpath.

The undirected version of Proposition 3.3.2 also holds. The proof is obtained by changing the word "path" to "chain".

Proposition 3.4.2 *Let G be any digraph. (a) For any two nodes, u, v, in G, every non-null chain, π, from u to v contains a simple non-null subchain.*

(b) If $|V| = n$, then every open simple chain has length at most $n-1$ and every closed simple chain has length at most n.

The undirected version of strong connectivity is the following:

Definition 3.4.3 Let $G = (V, E, s, t)$ be a digraph. We define the binary relation, $\widetilde{C}_G$, on V as follows: For all $u, v \in V$,

$$u\widetilde{C}_G v \quad \text{iff} \quad \text{there is a chain from } u \text{ to } v.$$

When $u\widetilde{C}_G v$, we say that *u and v are connected*.

Oberve that the relation $\widetilde{C}_G$ is an equivalence relation.

The equivalence classes of the relation $\widetilde{C}_G$ are called the *connected components of G (CC's)*.

A graph is *connected* iff it has a single connected component.

Observe that strong connectivity implies connectively but the converse is false.

For example, the graph G_1 of Figure 3.4 is connected but it is not strongly connected.

The function *strcomp* and the method for computing the strongly connected components of a graph can easily be adapted to compute the connected components of a graph.

The undirected version of a circuit is the following:

Definition 3.4.4 Let $G = (V, E, s, t)$ be a digraph. A *cycle* is a closed e-simple chain (*i.e.*, no edge occurs twice) and a *simple cycle* is a simple closed chain. The null chain, (u, ϵ, u), is a simple cycle.

Remark: A closed chain is sometimes called a *pseudo-cycle*. The undirected version of Proposition 3.3.6 also holds.

Again, the proof consists in changing the word "circuit" to "cycle".

Proposition 3.4.5 *Let G be any digraph. (a) Every cycle, π, in G is the concatenation of pairwise edge-disjoint simple cycles.*

(b) A cycle is simple iff it is a minimal cycle, that is, iff it does not contain any proper cycle.

The reader should now be convinced that it is actually possible to use the notion of a directed graph to model a large class of problems where the notion of orientation is irrelevant.

However, this is somewhat unnatural and often inconvenient, so it is desirable to introduce the notion of an undirected graph as a "first-class" object. How should we do that?

We could redefine the set of edges of an undirected graph to be of the form $E^+ \cup E^-$, where $E^+ = E$ is the original set of edges of a digraph and with

$$E^- = \{e^- \mid e^+ \in E^+,\ s(e^-) = t(e^+),\ t(e^-) = s(e^+)\},$$

each edge, e^-, being the "anti-edge" (opposite edge) of e^+.

Such an approach is workable but experience shows that it not very satisfactory.

The solution adopted by most people is to relax the condition that every edge, $e \in E$, is assigned an *ordered pair*, $\langle u, v\rangle$, of nodes (with $u = s(e)$ and $v = t(e)$) to the condition that every edge, $e \in E$, is assigned a *set*, $\{u, v\}$ of nodes (with $u = v$ allowed).

To this effect, let $[V]^2$ denote the subset of the power set consisting of all two-element subsets of V (the notation $\binom{V}{2}$ is sometimes used instead of $[V]^2$) :

$$[V]^2 = \{\{u, v\} \in 2^V \mid u \neq v\}.$$

Definition 3.4.6 A *graph* is a triple, $G = (V, E, st)$, where V is a set of *nodes or vertices*, E is a set of *arcs or edges* and $st\colon E \to V \cup [V]^2$ is a function that assigns a set of *endpoints* (or *endnodes*) to every edge.

When we want to stress that we are dealing with an undirected graph as opposed to a digraph, we use the locution *undirected graph*.

When we draw an undirected graph we suppress the tip on the extremity of an arc.

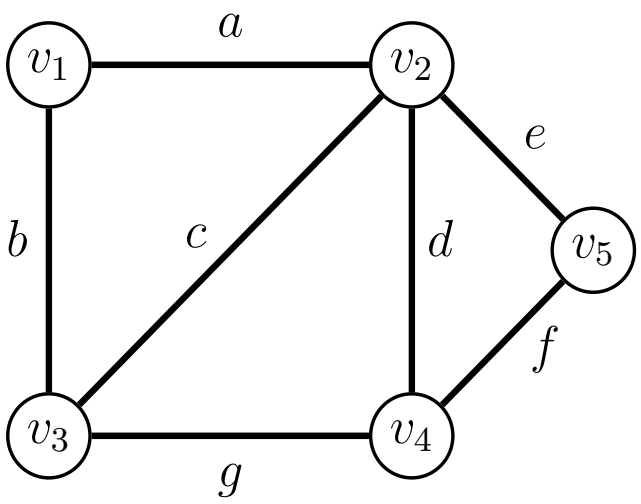

Figure 3.16: The Undirected Graph G_6

For example, the undirected graph, G_6, corresponding to the directed graph G_5 is shown in Figure 3.16.

Definition 3.4.7 Given a graph, G, an edge, $e \in E$, such that $st(e) \in V$ is called a *loop* (or *self-loop*). Two edges, $e, e' \in E$ are said to be *parallel edges* iff $st(e) = st(e')$. A graph is *simple* iff it has no loops and no parallel edges.

Remarks:

1. The functions st need not be injective or surjective.
2. When G is simple, every edge, $e \in E$, is uniquely determined by the set of vertices, $\{u, v\}$, such that $\{u, v\} = st(e)$.

   In this case, we may denote the edge e by $\{u, v\}$ (some books also use the notation (uv) or even uv).
3. Some authors call a graph with no loops but possibly parallel edges a *multigraph* and a graph with loops and parallel edges a *pseudograph*. We prefer to use the term graph for the most general concept.
4. Given an undirected graph, $G = (V, E, st)$, we can form directed graphs from G by assigning an arbitrary orientation to the edges of G.

This means that we assign to every set, $st(e) = \{u, v\}$, where $u \neq v$, one of the two pairs (u, v) or (v, u) and define s and t such that $s(e) = u$ and $t(e) = v$ in the first case or such that $s(e) = v$ and $t(e) = u$ in the second case (when $u = v$, we have $s(e) = t(e) = u$)).

5. When a graph is simple, the function st is often omitted and we simply write (V, E), with the understanding that E is a set of two-elements subsets of V.

6. The concepts or adjacency and incidence transfer immediately to (undirected) graphs.

It is clear that the Definition of chain, connectivity, and cycle (Definitions 3.4.1, 3.4.3 and 3.4.4) immediately apply to (undirected) graphs.

For example, the notion of a chain in an undirected graph is defined as follows:

Definition 3.4.8 Given any graph, $G = (V, E, st)$, and any two nodes, $u, v \in V$, a *chain* (or *walk*) *from u to v* is a sequence $\pi = (u_0, e_1, u_1, e_2, u_2, \ldots, u_{n-1}, e_n, u_n)$, where $n \geq 1$; $u_i \in V$; $e_i \in E$ and

$$u_0 = u;\ u_n = v \quad \text{and} \quad st(e_i) = \{u_{i-1}, u_i\}, \quad 1 \leq i \leq n.$$

We call n the *length of the chain π* and we write $|\pi| = n$. When $n = 0$, we have the *null chain*, (u, ϵ, u), from u to u, a chain of length 0.

If $u = v$, then π is called a *closed chain*, else an *open chain*. The chain, π, determines the sequence of nodes, $\text{nodes}(\pi) = \langle u_0, \ldots, u_n \rangle$, with $\text{nodes}((u, \epsilon, u)) = \langle u, u \rangle$.

A chain, π, is *edge-simple*, for short, *e-simple* iff $e_i \neq e_j$ for all $i \neq j$ (*i.e.*, no edge in the chain is used twice).

A chain, π, from u to v is *simple* iff no vertex in $\text{nodes}(\pi)$ occurs twice, except possibly for u if π is closed. The null chain, (u, ϵ, u), is considered e-simple and simple.

An e-simple chain is also called a *trail* (as in the case of directed graphs).

Definitions 3.4.3 and 3.4.4 are adapted to undirected graphs in a similar fashion.

However, only the notion of *degree* (or *valency*) of a node applies to undirected graph where it is given by

$$d_G(u) = |\{e \in E \mid u \in st(e)\}|.$$

We can check immediately that Corollary 3.2.5 and Corollary 3.2.6 apply to undirected graphs.

Remark: When it is clear that we are dealing with undirected graphs, we will sometimes allow ourselves some abuse of language. For example, we will occasionally use the term path instead of chain.

The notion of homomorphism and isomorphism also makes sense for undirected graphs.

In order to adapt Definition 3.2.7, observe that any function, $g\colon V_1 \to V_2$, can be extended in a natural way to a function from $V_1 \cup [V_1]^2$ to $V_2 \cup [V_2]^2$, also denoted g, so that

$$g(\{u, v\}) = \{g(u), g(v)\},$$

for all $\{u, v\} \in [V_1]^2$.

Definition 3.4.9 Given two graphs, $G_1 = (V_1, E_1, st_1)$ and $G_2 = (V_2, E_2, st_2)$, a *homomorphism* (or *morphism*), $f : G_1 \to G_2$, *from G_1 to G_2* is a pair, $f = (f^v, f^e)$, with $f^v : V_1 \to V_2$ and $f^e : E_1 \to E_2$, preserving incidence, that is, for every edge, $e \in E_1$, we have

$$st_2(f^e(e)) = f^v(st_1(e)).$$

These conditions can also be expressed by saying that the following diagram commute:

$$\begin{array}{ccc} E_1 & \xrightarrow{f^e} & E_2 \\ {\scriptstyle st_1}\downarrow & & \downarrow{\scriptstyle st_2} \\ V_1 \cup [V_1]^2 & \xrightarrow[f^v]{} & V_2 \cup [V_2]^2. \end{array}$$

As for directed graphs, we can compose homomorphisms of undirected graphs and the definition of an isomorphism of undirected graphs is the same as the definition of an isomorphism of digraphs.

Definition 3.3.9 about various notions of subgraphs is immediately adapted to undirected graphs.

We are now going to investigate the properties of a very important subclass of graphs, trees.

3.5 Trees and Arborescences

In this section, until further notice, we will be dealing with undirected graphs. Given a graph, G, edges having the property that their deletion increases the number of connected components of G play an important role and we would like to characterize such edges.

Definition 3.5.1 Given any graph, $G = (V, E, st)$, any edge, $e \in E$, whose deletion increases the number of connected components of G (*i.e.*,
$(V, E - \{e\}, st \restriction (E - \{e\}))$ has more connected components than G) is called a *bridge*.

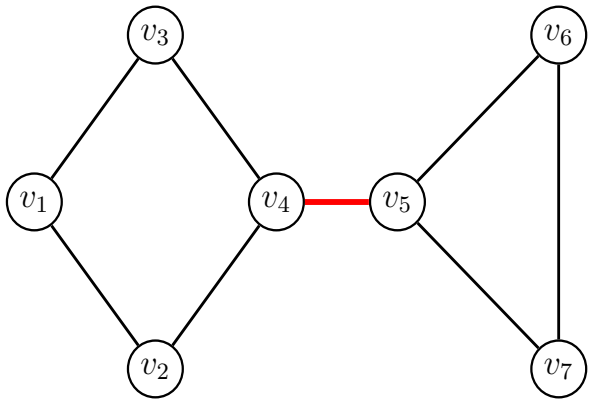

Figure 3.17: A bridge in the graph G_7

The edge (v_4v_5) is a bridge (see Figure 3.17).

Proposition 3.5.2 *Given any graph, $G = (V, E, st)$, adjunction of a new edge, e, between u and v (this means that st is extended to st_e, with $st_e(e) = \{u, v\}$) to G has the following effect:*

1. *Either the number of components of G decreases by 1, in which case the edge e does not belong to any cycle of $G' = (V, E \cup \{e\}, st_e)$, or*
2. *The number of components of G is unchanged, in which case the edge e belongs to some cycle of $G' = (V, E \cup \{e\}, st_e)$.*

Corollary 3.5.3 *Given any graph, $G = (V, E, st)$, an edge, $e \in E$, is a bridge iff it does not belong to any cycle of G.*

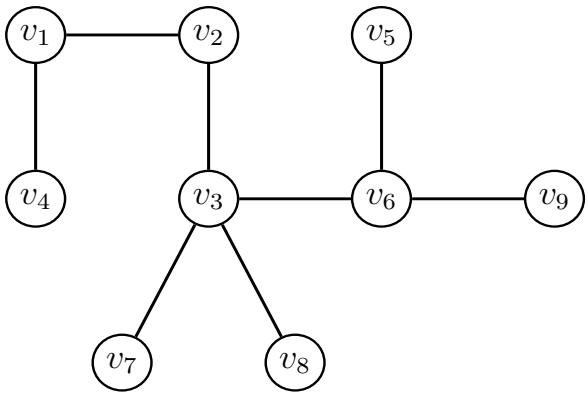

Figure 3.18: A Tree, T_1

Theorem 3.5.4 *Let G be a finite graph and let $m = |V| \geq 1$. The following properties hold:*

(i) If G is connected, then $|E| \geq m - 1$.

(ii) If G has no cycle, then $|E| \leq m - 1$.

In view of Theorem 3.5.4, it makes sense to define the following kind of graphs:

Definition 3.5.5 A *tree* is a graph that is connected and acyclic (*i.e.*, has no cycles). A *forest* is a graph whose connected components are trees.

The picture of a tree is shown in Figure 3.18.

Our next theorem gives several equivalent characterizations of a tree.

Theorem 3.5.6 *Let G be a finite graph with $m = |V| \geq 2$ nodes. The following properties characterize trees:*

(1) G is connected and acyclic.

(2) G is connected and minimal for this property (if we delete any edge of G, then the resulting graph is no longer connected).

(3) G is connected and has $m - 1$ edges.

(4) G is acyclic and maximal for this property (if we add any edge to G, then the resulting graph is no longer acyclic).

(5) G is acyclic and has $m - 1$ edges.

(6) Any two nodes of G are joined by a unique chain.

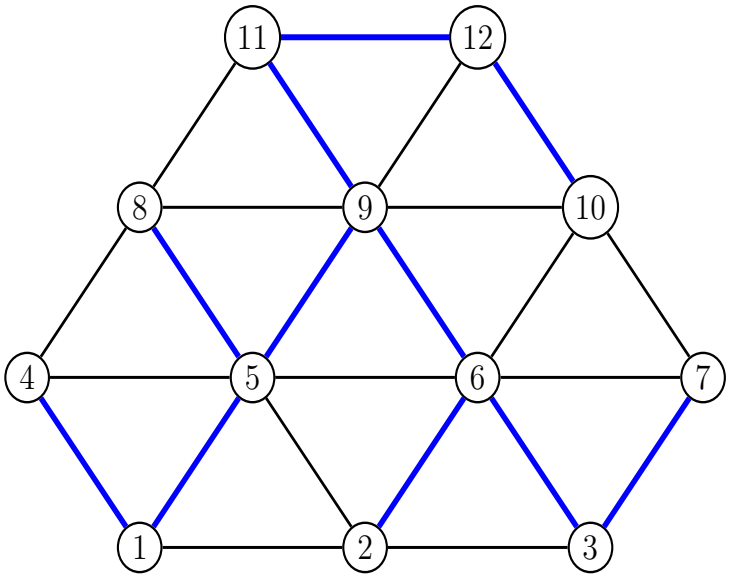

Figure 3.19: A Spanning Tree

Remark: The equivalence of (1) and (6) holds for infinite graphs too.

Corollary 3.5.7 *For any tree, G, adding a new edge, e, to G yields a graph, G', with a unique cycle.*

Corollary 3.5.8 *Every finite connected graph possesses a spanning tree.*

An example of a spanning tree (shown in thicker lines) in a graph is shown in Figure 3.19.

An *endpoint* or *leaf* in a graph is a node of degree 1.

Proposition 3.5.9 *Every finite tree with $m \geq 2$ nodes has at least two endpoints.*

Remark: A forest with m nodes and p connected components has $m - p$ edges. Indeed, if each connected component has m_i nodes, then the total number of edges is

$$(m_1 - 1) + (m_2 - 1) + \cdots + (m_p - 1) = m - p.$$

We now consider briefly directed versions of a tree.

Definition 3.5.10 Given a digraph, $G = (V, E, s, t)$, a node, $a \in V$ is a *root* (resp. *anti-root*) iff for every node $u \in V$, there is a path from a to u (resp. there is a path from u to a). A digraph with at least two nodes is an *arborescence with root a* iff

1. The node a is a root of G
2. G is a tree (as an undirected graph).

A digraph with at least two nodes is an *anti-arborescence with anti-root a* iff

1. The node a is an anti-root of G
2. G is a tree (as an undirected graph).

Note that orienting the edges in a tree does not necessarily yield an arborescence (or an anti-arborescence).

Also, if we reverse the orientation of the arcs of an arborescence we get an anti-arborescence.

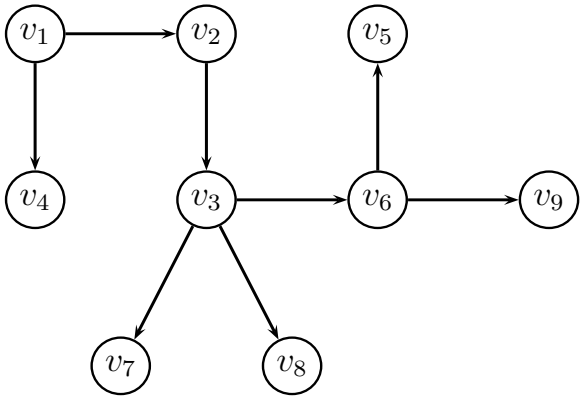

Figure 3.20: An Arborescence, T_2

An arborescence is shown is Figure 3.20.

There is a version of Theorem 3.5.6 giving several equivalent characterizations of an arborescence.

Theorem 3.5.11 *Let G be a finite digraph with $m = |V| \geq 2$ nodes. The following properties characterize arborescences with root a:*

(1) G is a tree (as undirected graph) with root a.

(2) For every $u \in V$, there is a unique path from a to u.

(3) G has a as a root and is minimal for this property (if we delete any edge of G, then a is not a root any longer).

(4) G is connected (as undirected graph) and moreover

$$(*) \begin{cases} d^-_G(a) = 0 \\ d^-_G(u) = 1, \text{ for all } u \in V,\ u \neq a. \end{cases}$$

(5) G is acyclic (as undirected graph) and the properties $(*)$ are satisfied.

(6) G is acyclic (as undirected graph) and has a as a root.

(7) G has a as a root and has $m - 1$ arcs.

3.6 Minimum (or Maximum) Weight Spanning Trees

For a certain class of problems, it is necessary to consider undirected graphs (without loops) whose edges are assigned a "cost" or "weight".

Definition 3.6.1 A *weighted graph* is a finite graph without loops, $G = (V, E, st)$, together with a function, $c\colon E \to \mathbb{R}$, called a *weight function* (or *cost function*). We will denote a weighted graph by (G, c). Given any set of edges, $E' \subseteq E$, we define the *weight (or cost)* of E' by

$$c(E') = \sum_{e \in E'} c(e).$$

Given a weighted graph, (G, c), an important problem is to find a spanning tree, T such that $c(T)$ is maximum (or minimum).

This problem is called the *maximal weight spanning tree* (resp. *minimal weight spanning tree*).

Actually, it is easy to see that any algorithm solving any one of the two problems can be converted to an algorithm solving the other problem.

For example, if we can solve the maximal weight spanning tree, we can solve the minimal weight spanning tree by replacing every weight, $c(e)$, by $-c(e)$, and by looking for a spanning tree, T, that is a maximal spanning tree, since

$$\min_{T \subseteq G} c(T) = -\max_{T \subseteq G} -c(T).$$

There are several algorithms for finding such spanning trees, including one due to Kruskal and another one due to Prim.

The fastest known algorithm at the present is due to Bernard Chazelle (1999).

Since every spannning tree of a given graph, $G = (V, E, st)$, has the same number of edges (namely, $|V| - 1$), adding the same constant to the weight of every edge does not affect the maximal nature a spanning tree, that is, the set of maximal weight spanning trees is preserved.

Therefore, *we may assume that all the weights are non-negative*.

In order to justify the correctness of Kruskal's algorithm, we need two definitions.

Let (G, c) be any connected weighted graph with $G = (V, E, st)$ and let T be any spanning tree of G.

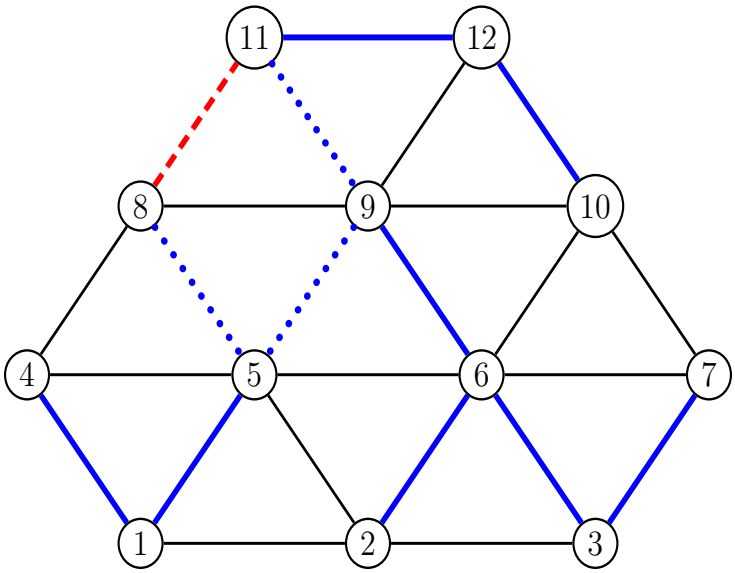

Figure 3.21: The set C_e associated with an edge $e \in G - T$

For every edge, $e \in E - T$, let C_e be the set of edges belonging to the unique chain in T joining the endpoints of e (the vertices in $st(e)$).

For example, in the graph shown in Figure 3.21, the set $C_{\{8,11\}}$ associated with the edge $\{8, 11\}$ (shown as a dashed line) corresponds to the following set of edges (shown as dotted lines) in T:

$$C_{\{8,11\}} = \{\{8, 5\}, \{5, 9\}, \{9, 11\}\}.$$

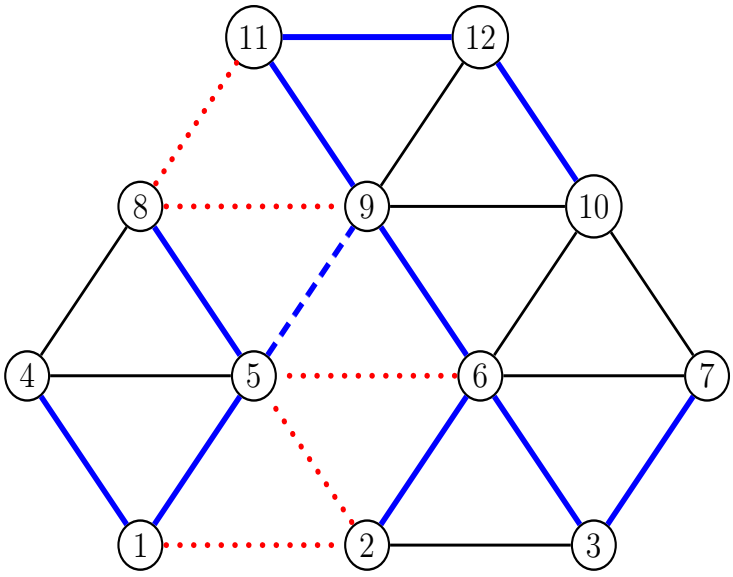

Figure 3.22: The set $\Omega_{\{5,9\}}$ obtained by deleting the edge $\{5,9\}$ from the spanning tree.

Also, given any edge, $e \in T$, observe that the result of deleting e yields a graph denoted $T - e$ consisting of two disjoint subtrees of T.

We let Ω_e be the set of edges, $e' \in G - T$, such that if $st(e') = \{u, v\}$, then u and v belong to the two distinct connected components of $T - \{e\}$.

For example, in Figure 3.22, deleting the edge $\{5,9\}$ yields the set of edges (shown as dotted lines)

$$\Omega_{\{5,9\}} = \{\{1,2\},\{5,2\},\{5,6\},\{8,9\},\{8,11\}\}.$$

Observe that in the first case, deleting any edge from C_e and adding the edge $e \in E - T$ yields a new spanning tree and in the second case, deleting any edge $e \in T$ and adding any edge in Ω_e also yields a new spanning tree.

These observations are crucial ingredients in the proof of the following theorem:

Theorem 3.6.2 *Let (G, c) be any connected weighted graph and let T be any spanning tree of G.*

(1) The tree T is a maximal weight spanning tree iff any of the following (equivalent) conditions hold:

(i) For every $e \in E - T$,

$$c(e) \leq \min_{e' \in C_e} c(e')$$

(ii) For every $e \in T$,

$$c(e) \geq \max_{e' \in \Omega_e} c(e').$$

Figure 3.23: Joseph Kruskal, 1928-

(2) The tree T is a minimal weight spanning tree iff any of the following (equivalent) conditions hold:

(i) For every $e \in E - T$,

$$c(e) \geq \max_{e' \in C_e} c(e')$$

(ii) For every $e \in T$,

$$c(e) \leq \min_{e' \in \Omega_e} c(e').$$

We are now in the position to present a version of Kruskal's algorithm and to prove its correctness.

Here is a version of Kruskal's algorithm for finding a minimal weight spanning tree using criterion 2(i).

Let n be the number of edges of the weighted graph, (G, c), where $G = (V, E, st)$.

function *Kruskal*((G, c): weighted graph): tree
  begin
    Sort the edges in non-decreasing order of weights:
    $c(e_1) \leq c(e_2) \leq \cdots \leq c(e_n)$;
    $T := \emptyset$;
    for $i := 1$ **to** n **do**
      if $(V, T \cup \{e_i\})$ is acyclic **then** $T := T \cup \{e_i\}$
      endif
    endfor;
    Kruskal $:= T$
  end

We admit that the above description of Kruskal's algorithm is a bit sketchy as we have not explicitly specified how we check that adding an edge to a tree preserves acyclicity.

On the other hand, it is quite easy to prove the correctness of the above algorithm.

It is not difficult to refine the above "naive" algorithm to make it totally explicit but this involves a good choice of data structures. We leave these considerations to an algorithms course.

Clearly, the graph T returned by the algorithm is acyclic, but why is it connected? This can be proved too.

We can easily design a version of Kruskal's algorithm based on condition 2(ii).

This time, we sort the edges in non-increasing order of weights and, starting with G, we attempt to delete each edge, e_j, as long as the remaining graph is still connected.

Prim's algorithm is based on a rather different observation.

For any node, $v \in V$, let U_v be the set of edges incident with v that are not loops,

$$U_v = \{e \in E \mid v \in st(e),\ st(e) \in [V]^2\}.$$

Choose in U_v some edge of minimum weight which we will (ambiguously) denote by $e(v)$.

Proposition 3.6.3 *Let (G, c) be a connected weighted graph with $G = (V, E, st)$. For every vertex, $v \in V$, there is a minimum weight spanning tree, T, so that $e(v) \in T$.*

Prim's algorithm uses an edge-contraction operation described below:

Definition 3.6.4 Let $G = (V, E, st)$ be a graph, and let $e \in E$ be some edge which is not a loop, *i.e.*, $st(e) = \{u, v\}$, with $u \neq v$. The graph, $C_e(G)$, obtained by *contracting the edge e* is the graph obtained by merging u and v into a single node and deleting e. More precisely, $C_e(G) = ((V - \{u, v\}) \cup \{w\}, E - \{e\}, st_e)$, where w is any new node not in V and where

1. $st_e(e') = st(e')$ iff $u \notin st(e')$ and $v \notin st(e')$
2. $st_e(e') = \{w, z\}$ iff $st(e') = \{u, z\}$, with $z \notin st(e)$
3. $st_e(e') = \{z, w\}$ iff $st(e') = \{z, v\}$, with $z \notin st(e)$
4. $st_e(e') = w$ iff $st(e') = \{u, v\}$.

Proposition 3.6.5 *Let $G = (V, E, st)$ be a graph. For any edge, $e \in E$, the graph G is a tree iff $C_e(G)$ is a tree.*

Here is a "naive" version of Prim's algorithm.

function *Prim*$((G = (V, E, st), c)$: weighted graph): tree
  begin
    $T := \emptyset$;
    while $|V| \geq 2$ **do**
      pick any vertex $v \in V$;
      pick any edge (not a loop), e, in U_v of minimum weight;
      $T := T \cup \{e\}$; $G := C_e(G)$
    endwhile;
    Prim $:= T$
  end

The correctness of Prim's algorithm is an immediate consequence of Proposition 3.6.3 and Proposition 3.6.5, the details are left to the reader.