Chapter 7

Path problems

7.1 Finding Short Paths and Long Paths

7.1.1 The Shortest Path Algorithm

Theorem 7.1 (Shortest Path Algorithm)**.** *The shortest path from a vertex a to a vertex z in a connected weighted graph G can be found by labelling vertices of the graph in an appropriate manner, then using the labelling to construct the path. Each vertex is initially unlabelled; as the labelling algorithm proceeds we assign temporary labels $T(u)$ or permanent labels $P(u)$ to vertices u of G.*

The labelling is carried out as follows:

1. *Assign the permanent label 0 to a, so $P(a) = 0$.*

2. *Suppose u is the vertex that has most recently been assigned a permanent label, $P(u)$. For each vertex x adjacent to u let $w_{\min}(ux)$ denote the minimum weight of any edge that joins u and x. If x is unlabelled, or if $P(u)+w_{\min}(ux) < T(x)$ then set $T(x) = P(u)+w_{\min}(ux)$. Otherwise, leave $T(x)$ unchanged.*

3. *Select a vertex u with the smallest temporary label $T(u)$ (if there is more than one you can choose arbitrarily), and assign to it the permanent label $P(u) = T(u)$.*

4. *If z has a permanent label $P(z)$ the labelling process terminates, and $P(z)$ is the length of the shortest path from a to z. Otherwise, the process is repeated from step 2.*

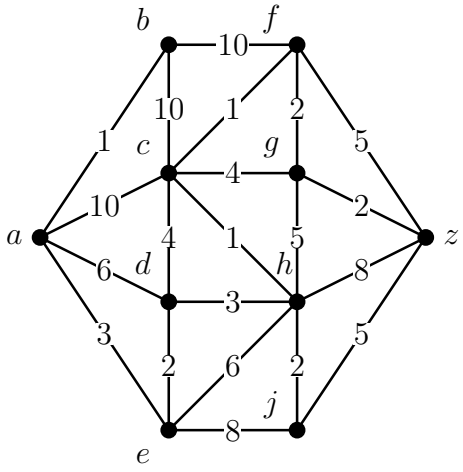
Once the labelling is completed, we recover the shortest path as follows:

1. *Starting from z, find a vertex u adjacent to z such that $P(u) + w_{\min}(uz) = P(z)$.*

2. *Repeat for vertex u instead of z, and continue until vertex a is reached.*

We note that this algorithm can also be applied to networks (weighted digraphs) by modifying step 2 so that we consider only vertices x for which there exists an arc from u to x.

Example 7.1. Find the shortest path between vertices a and z in the following graph:



We carry out the labelling process as follows:

Iteration	Permanent Labels	Temporary Labels
1	P(0)=0	
2	P(0)=0,P(b)=1	T(b)=1,T(c)=10,T(d)=6,T(e)=3
3	P(0)=0,P(b)=1, P(e)=3	T(c)=10,T(d)=6,T(e)=3,T(f)=11
4	P(0)=0,P(b)=1, P(e)=3, P(d)=5	T(c)=10, T(d)=5, T(f)=11, T(h)=9, T(j)=11
5	P(0)=0,P(b)=1, P(e)=3, P(d)=5, P(h)=8	T(c)=9, T(f)=11, T(h)=8, T(j)=11
6	P(0)=0,P(b)=1, P(e)=3, P(d)=5, P(h)=8, P(c)=9	T(c)=9, T(f)=11, T(g)=13, T(j)=10, T(z)=16
7	P(0)=0,P(b)=1, P(e)=3, P(d)=5, P(h)=8, P(c)=9, P(f)=10	T(f)=10, T(g)=13, T(j)=10, T(z)=16
8	P(0)=0,P(b)=1, P(e)=3, P(d)=5, P(h)=8, P(c)=9, P(f)=10, P(j)=10	T(g)=12, T(j)=10, T(z)=15
9	P(0)=0,P(b)=1, P(e)=3, P(d)=5, P(h)=8, P(c)=9, P(f)=10, P(j)=10, P(g)=12	T(g)=12, T(z)=15
10	P(0)=0,P(b)=1, P(e)=3, P(d)=5, P(h)=8, P(c)=9, P(f)=10, P(j)=10, P(g)=12, P(z)=14	T(z)=14

Alternatively, to keep track of the labels in a more compact form we could write the temporary labels at each step into a table as follows, with a label being marked in bold once it is converted

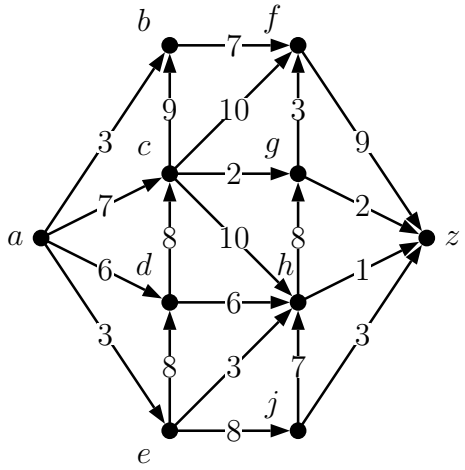
into a permanent label.

a	b	c	d	e	f	g	h	j	z
0									
	1	10	6	3					
		10	6	**3**	11				
		10	**5**		11		9	11	
		9			11		**8**	11	
		9			11	13		10	16
					10	13		10	16
						12		**10**	15
						12			15
									14

Having completed the labelling, we conclude that the length of the shortest path from a to z is 14. In order to find the path, we start by examining the vertices adjacent to z. We observe that g satisfies $P(g) + w_{\min}(gz) = 12 + 2 = P(z)$, hence the final edge of our path is gz. We now consider the neighbours of g, and observe that $P(f) + w_{\min}(fg) = 10 + 2 = P(g)$, so we know we can end our shortest path by f, g, z. Considering neighbours of f we note that $P(c)+w_{\min}(cf) = 9+1 = P(f)$, so we can end our path by c, f, g, z. Continuing in this manner, we can eventually determine that a, e, d, c, f, g, z is a path of length 14 from a to z.

We observe that whenever a vertex v is assigned a permanent label $P(v)$, then $P(v)$ is the length of the shortest path from a to v.

Exercise 7.2. Find the shortest path between vertices a and z in the following network:



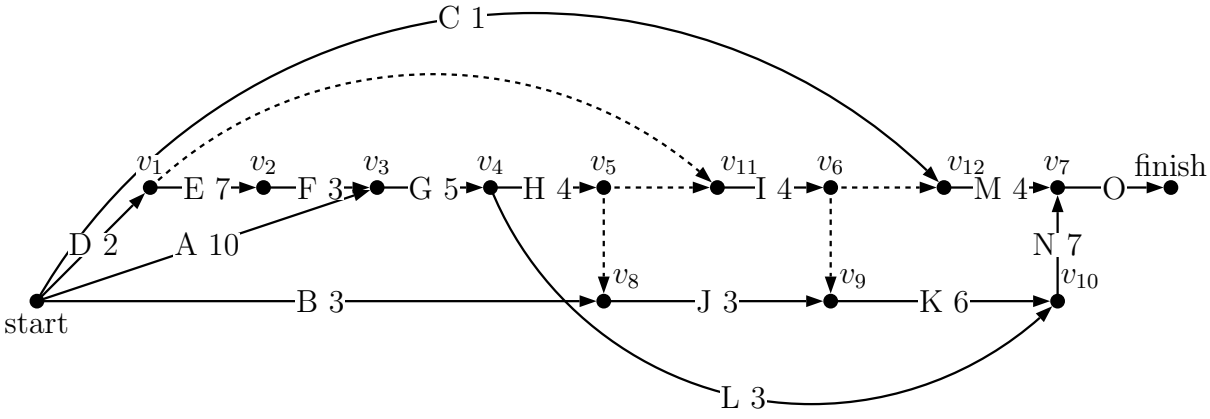
7.1.2 Scheduling Problems and The Longest Path Algorithm

There are several possible motivations for finding short paths in a graph or network; it is perhaps not immediately clear, however, why we might wish to find a long path. One application of such a technique is in what is known as *critical path analysis*, which relates to the problem of how to schedule the individual tasks involved in a complex project. In such a case, some of the tasks may have to wait for other tasks to be completed (you can't ice a cake before you have baked it!) whereas others can be undertaken simultaneously. In order to determine the minimum amount of time required to complete a project, it is necessary to find the longest sequence of tasks that must be carried out. In order to do this, we represent the tasks as an *activity network*, with each task corresponding to an arc that is assigned a weight equal to the time taken to complete the task. The arcs going in to a vertex of the network represent tasks that must be completed before the tasks corresponding to arcs going out of the vertex can begin.

Example 7.3. The Chew-Chew Restaurant Company wish to convert old railway carriages into restaurants. The activities involved in the conversion are as shown.

Activity	Description	Pre-requisites	Duration (weeks)
A	Purchase and renovate coaches	–	10
B	Purchase restaurant equipment	–	3
C	Hire personnel	–	1
D	Select and purchase site	–	2
E	Obtain licences	D	7
F	Site preparation	E	3
G	Move coaches to site	A, F	5
H	Install utilities	G	4
I	Install equipment	D, H	4
J	Decorate	B, H	3
K	Stock bar and kitchen	I, J	6
L	Advertise	G	3
M	Train staff	C, I	4
N	Pilot operation	K, L	7
O	Start full operation	M, N	–

This project can be represented by the following activity network:



The dashed arcs are known as *dummy arcs*. They do not represent tasks, and as such have no weight associated with them. However, they are sometimes necessary for the purpose of indicating dependencies between tasks in the case where a given task is a pre-requisite for more than one other task. In the above network, for example, they are used to indicate that task H is a pre-requisite for both task I and task J.

There may be more than one possible way to draw the network, but they will be equivalent in terms of the information they represent. We observe that an activity network cannot contain a directed cycle, as this would represent a sequence of tasks that were all dependent on each other and hence could not ever be started.

Definition 7.1. The longest path from the start vertex to the finish vertex in an activity network is known as the *critical path*. The length of the critical path represents the minimum amount of time that is necessary for the completion of the corresponding project.

We can find the critical path by means of a *longest path algorithm* (note that this algorithm only works for networks without directed cycles).

Theorem 7.2 (Longest Path Algorithm)**.** *The longest path from a vertex a to a vertex z in a connected network N with no directed cycles can be found using a technique similar to that for finding a shortest path. We begin by labelling the vertices as follows:*

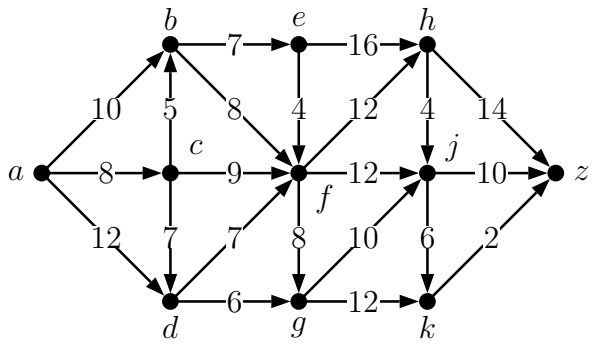
1. *Assign the permanent label 0 to a, so $P(a) = 0$. To each vertex v that can be reached only by an arc directly from a we assign a permanent label equal to the weight of that arc.*

2. *Consider all vertices that can be reached only by means of arcs directly from vertices that have permanent labels. For each such vertex x, consider each arc vx into $x$and assign the temporary label $T(x) = P(v) + \text{weight}(vx)$ to x, unless x already has a larger temporary label. When all arcs into x have been considered, set $P(x) = T(x)$.*

3. *Repeat step 2. until z has received a permanent label. This label is the length of the longest path from a to z.*

We can recover the longest path from this labelling in the same way that we constructed the shortest path in Theorem 7.1.

Example 7.4. Find the longest path from a to z in the following network.

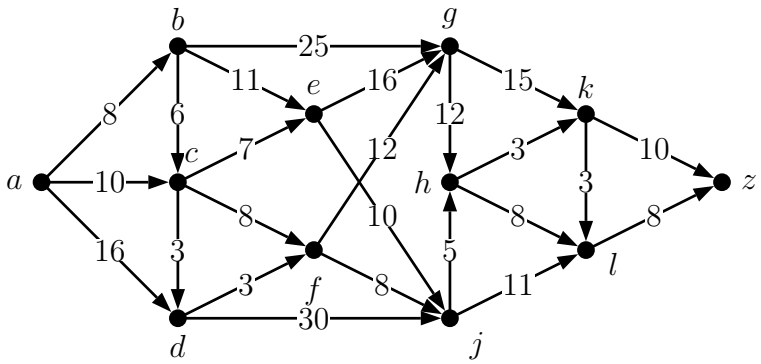


Step 1. $P(a) = 0$
$P(c) = 8$

Step 2. $P(b) = 13$
$P(d) = 15$
$P(e) = 20$
$P(f) = 24$
$P(g) = 32$
$P(h) = 36$
$P(j) = 42$
$P(k) = 48$
$P(z) = 52$

Step 3. To recover the longest path, we observe that $P(z) - P(j) = 52 - 42 = 10$, which is the weight of the arc from j to z, so the final arc in our path is jz. Similarly, we note that $P(j) - P(g) = 42 - 32 = 10$, which is the weight of gj, so the path ends gjz. Continuing in this manner, we see that the longest path is $acbefgjz$, and it has length $8 + 5 + 7 + 4 + 8 + 10 + 10 = 52 = P(z)$.

Exercise 7.5. Find the longest path in the following digraph:



Example 7.6. Returning to our example of the Chew-Chew Restaurant company, we apply

the longest path algorithm to find the longest path in the network:

$$\begin{aligned} P(\text{start}) &= 0 \\ P(v_1) &= 2 \\ P(v_2) &= 9 \\ P(v_3) &= 12 \\ P(v_4) &= 17 \\ P(v_5) &= 21 \\ P(v_{11}) &= 21 \\ P(v_6) &= 25 \\ P(v_8) &= 21 \\ P(v_9) &= 25 \\ P(v_{12}) &= 25 \\ P(v_{10}) &= 31 \\ P(v_7) &= 38 \\ P(\text{finish}) &= 38. \end{aligned}$$

Working backwards we can determine that the longest path is

$$\text{start} \to D \to E \to F \to G \to H \to I \to K \to N \to O \to \text{finish}.$$

This is the *critical path* for this activity network. The fact that it has length 38 means that the minimum time necessary to complete the project is 38 weeks.

This approach can be extended in order to obtain more-detailed information about the constraints inherent in trying to schedule the various tasks:

Activity	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Earliest Start	0	0	0	0	2	9	12	17	21	21	25	17	25	31	38
Latest Start	2	19	33	0	2	9	12	17	21	22	25	28	34	31	38
Critical?				✓	✓	✓	✓	✓	✓		✓			✓	✓
Float time	2	19	33	0	0	0	0	0	0	1	0	11	9	0	0

Theorem 7.3. *For a task represented in a network by an arc xy, the earliest possible start time is given by the length of the longest path from a to x, and the latest possible start time (if the project is to be completed in the minimum possible time) is equal to the difference between the length of the critical path and the length of the longest path from x to z that includes the arc xy.*

The *float time* is the difference between the earliest starting time and the latest starting time; it describes how much leeway there is in scheduling the start of a given activity. Tasks that are part of the critical path in the network are said to be *critical activities*, and they have a float time of 0. This means that the start time for these activities cannot be varied if the project is to be completed within the 38 weeks, and any delay in carrying out these activities will delay the completion of the project.

7.1.3 Constructing an Activity Network from a List of Tasks and Prerequisites

Having seen how to analyse an activity network, we need to consider how to construct an activity network from a given list of tasks and their prerequisites. A straightforward way to do this is as follows:

1. Draw each task on the page as an arc with a vertex at either end.

2. Whenever task X has a prerequisite task Y, add a dummy arc connecting the vertex at the end of the arc corresponding to Y to the vertex at the start of the arc corresponding to X.

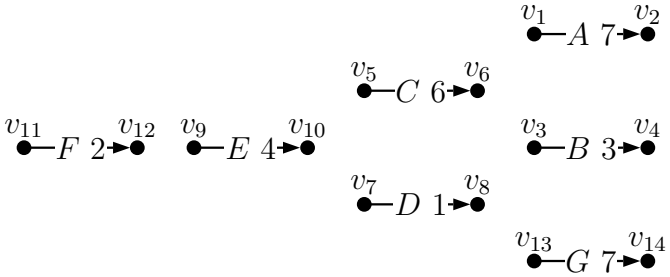
3. Add a 'start' vertex and for any tasks without prerequisites, add dummy arcs from the start vertex to the vertex at the start of those tasks.

4. Add a 'finish' vertex and for any tasks that are not prerequisites for any other tasks, add dummy arcs from the vertex at the end of those tasks to the finish vertex.

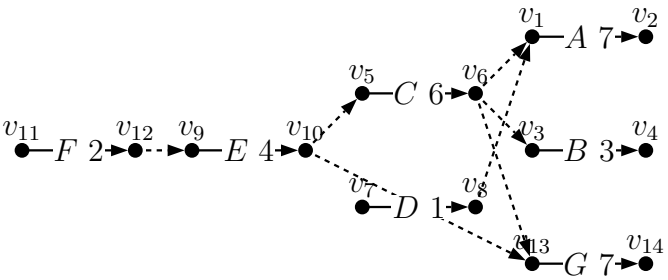
Example 7.7. Consider the following list of tasks and prerequisites:

Activity	Pre-requisites	Duration
A	C, D	7
B	C	3
C	E	6
D	–	1
E	F	4
F	–	2
G	E, C	7

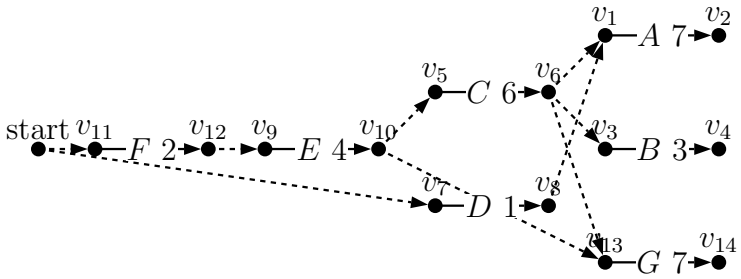
To construct an activity network we first draw arcs corresponding to each task:



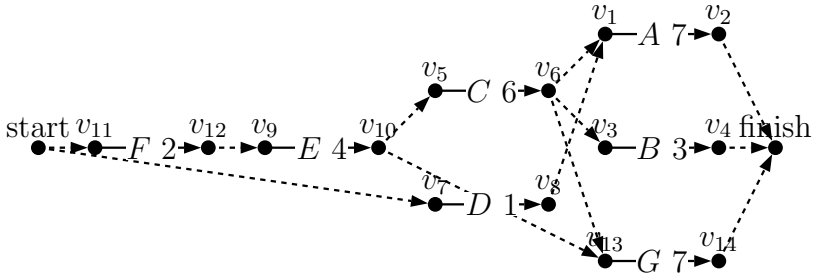
Next we add dummy arcs to indicate all prerequisites:



Add the start vertex and corresponding dummy arcs:



Complete the activity network by adding the finish vertex and corresponding dummy arcs:



Exercise 7.8. Suppose a project requires the following activities to be completed.

Activity	Pre-requisites	Duration
A	–	5
B	A	7
C	–	1
D	–	3
E	D	3
F	C, D	4
G	E	5
H	F	3
I	H, G	2
J	B, F	2
K	B	8
L	J, K	4
M	L, I	6

1. Draw an activity network corresponding to this project.
2. Use the longest path algorithm to find a critical path for this project.
3. Find the earliest start times for each of the activities in this project.
4. Find the latest start times for each of the activities in this project.
5. Determine the float time for each activity.

7.2 Paths and Connectedness

We are now going to look more closely at notions of *connectivity* in a graph. We have seen examples of graphs in which removing a single edge results in a disconnected graph; however, some other graphs remain connected even after several edges have been deleted. Considerations of this nature become important in the sort of application where a graph may represent an electricity network for a city, for example, and you would like to know how many power lines must fail before a blackout is caused in part of the city. We will begin formalising definitions of several concepts relating to connectivity, before going on to study how connectivity is closely related to the existence of paths between vertices of a graph.

7.2.1 Edge Connectivity

A tree is an example of a graph that is "only just connected" in some sense, as the removal of even a single edge results in a graph that is disconnected. Recall that a *bridge* is an edge whose removal causes a graph to be disconnected (thus every edge of a tree is a bridge). Not every graph contains a bridge. However, we can extend this notion to give a way of describing how "connected" a connected graph is:

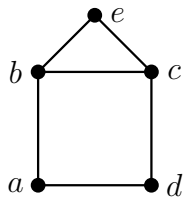
Definition 7.2. The *edge connectivity* of a connected graph G is the minimum number $\lambda(G)$ of edges that must be removed in order for the resulting graph to be disconnected.

Example 7.9. The edge connectivity of a tree T is $\lambda(T) = 1$. The edge connectivity of a cycle C_n (for $n \geq 3$) is $\lambda(C_n)$=2, since the removal of a single edge always results in a path (which is connected) but the removal of a second edge disconnects the path.

Definition 7.3. A connected graph G is said to be *k-edge connected* if $\lambda(G) \geq k$.

If a graph is k-edge connected, then any $k-1$ edges can be deleted and the resulting graph will still be connected.

Example 7.10. Consider the following graph G.



If any one edge is removed, the graph remains connected. However if edges ab and ad are removed, then the vertex a is disconnected from the rest of the graph. Thus the edge connectivity of G is 2, and G is 2-edge connected although it is not 3-edge connected.

Definition 7.4. A *disconnecting set* of a connected graph G is a set of edges of G whose removal disconnects G. A *cutset* of G is a disconnecting set C of G with the property that no proper subset of C is a disconnecting set.

Example 7.11. A bridge is an example of a disconnecting set that is also a cutset. If G is the graph from Example 7.10, then the set $\{ab, ad, bc\}$ is a disconnecting set, but it is not a cutset since it has a subset $\{ab, bd\}$ that is itself a disconnecting set. However, $\{ab, bd\}$ *is* a cutset, since it is a disconnecting set, but has no proper subset that is a disconnecting set. Note that the set $\{ec, bc, dc\}$ is also a cutset; this demonstrates that not every cutset has the same size. The edge connectivity of a graph gives the size of the smallest cutset.

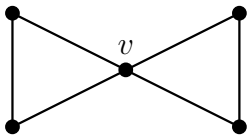
Exercise 7.12.
1. Find a cutset for the complete graph k_n, and use it to determine an upper bound on its k-edge connectivity. (Later we will see a theorem that will let us prove definitively that this is in fact the correct value.)
2. Considering the previous question, as well as Example 7.10, can you derive a relationship between vertex degrees in a graph and the edge-connectivity?

7.2.2 Vertex Connectivity

So far we have considered what happens to a graph when we delete edges. We can also consider deleting vertices (of course, when we delete a vertex we also have to delete all the edges adjacent to that vertex.)

Definition 7.5. A vertex v in a connected graph G is known as a *cut vertex* if its removal causes G to be disconnected.

Example 7.13.



The vertex v is a cut vertex for the above graph.

Definition 7.6. The *connectivity* of a connected graph G is the minimum number $\kappa(G)$ of vertices that must be removed in order for the resulting graph to be disconnected.

We note that this definition does not work in the special case of the complete graph K_n, as this graph always remains connected after any sequence of vertex deletions. By convention, we define $\kappa(K_n) = n - 1$.

Example 7.14. The connectivity of a tree T is $\kappa(T) = 1$. The connectivity of a cycle C_n (for $n \geq 3$) is $\kappa(C_n)$=2.

Definition 7.7. A graph G is said to be *k-connected* if $\kappa(G) \geq k$.

Example 7.15. Any graph with a cut vertex has connectivity 1. Hence if G is the graph of Example 7.13 then $\kappa(G) = 1$. However, $\lambda(G) = 2$. This is thus an example of a graph for which the edge connectivity is greater than the connectivity.

Definition 7.8. A set S of vertices of a connected graph G is a *separating set* of G if the deletion of the vertices in S causes the graph to be disconnected. A separating set S is a *vertex cutset* if S has no proper subset of vertices whose deletion cause the graph to be disconnected.

Theorem 7.4. *For a connected simple graph G, if $d_{\min}(G)$ is the smallest degree of any vertex in G then*

$$\kappa(G) \leq \lambda(G) \leq d_{\min}(G).$$

Proof. Let v be a vertex of G with degree $d_{\min}(G)$. By deleting all the edges incident with v we cause v to be disconnected from the rest of the graph, hence this set of $d_{\min}(G)$ edges is a disconnecting set, so we conclude $\lambda(G) \leq d_{\min}(G)$.

To demonstrate that the second inequality holds, we first observe that for a complete graph K_n we have $\kappa(K_n) = \lambda(K_n) = n-1$. Now suppose that G is not a complete graph. If G has edge connectivity $\lambda(G)$, then it has some cutset S of size $\lambda(G)$ such that $G \setminus S$ has two components. It is possible to find a set V of vertices that contains precisely one vertex incident with each edge in S having the property that V does not contain all the vertices in either component of $G \setminus S$. (If each component has more than one vertex, simply ensure V contains at least one vertex from each component. If one of the components is a single vertex, ensure all the vertices in V are chosen from the other component. Note that as G has edge connectivity $\lambda(G)$, the number of vertices in G is at least $\lambda(G) + 1$. Furthermore, as G is not a complete graph, it follows from the argument of the paragraph above that G has at least $\lambda(G) + 2$ vertices. This implies that the larger component in this case has size at least $\lambda(G) + 1$, and hence the vertices of V are a proper subset of those in the larger component.)

Then the size of V is at most $\lambda(G)$, and deleting the vertices in V will disconnect G (as it also leads to the deletion of all the edges in the cutset S,) hence V is a separating set for G. This implies that $\kappa(G) \leq \lambda(G)$. □

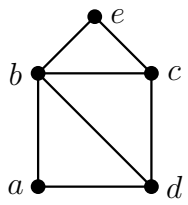
7.2.3 Vertex and Edge-Disjoint Paths Between Two Vertices

Have seen definitions for the connectivity and edge connectivity of a graph, we will now explore how these concepts relate to the existence of paths in a graph.

Example 7.16. Consider the graph of Example 7.13. We have noted that the vertex v is a cut vertex, whose deletion causes the graph to be disconnected. We further observe that any path from one of the two leftmost vertices to one of the two rightmost vertices of the graph must necessarily include v. Hence if v is deleted then there are no more paths from the leftmost vertices of v to the rightmost vertices of v (and thus the graph is disconnected by definition.)

Definition 7.9. If v and w are vertices of a connected graph G, then a path from v to w is referred to as a vw-path. Two vw-paths are said to be *edge disjoint* if they have no edges in common, and *vertex disjoint* if they have no vertices in common (apart from v and w).

Example 7.17. Let G be the graph



The paths $dabec$ and dbc are dc-paths; they are edge disjoint, but not vertex disjoint. The paths abc and adc are vertex disjoint (and edge disjoint) ac-paths.

Definition 7.10. Let v and w be vertices of a connected graph G. A set S of edges of G is said to be a vw-disconnecting set if every path from v to w contains an edge from S (in which case the deletion of the edges in S leaves a graph in which there are no longer any paths between v and w.) Similarly a set S of vertices is said to be a vw-separating set if $v, w \notin S$ and any path from v to w contains a vertex from S.

We note that any vw-disconnecting set is a disconnecting set of G, and any vw-separating set is a separating set.

Example 7.18. In the graph of Example 7.17, the set of edges $\{ab, bd, dc\}$ separates a from e. The set b, d of vertices separates a from c.

We are now in a position to give a theorem relating vw-paths and vw-disconnecting sets.

Theorem 7.5 (Menger's Theorem: Edge Form)**.** *If v and w are two vertices of a connected graph G, then the maximum number of edge-disjoint vw-paths in G is equal to the size k of the smallest vw-disconnecting set.*

By the definition of a vw-disconnecting set, it follows immediately that the maximum number of edge-disjoint vw paths is no greater than the size of the smallest vw-disconnecting set. For, if S is a vw-disconnecting set, then every vw-path contains some edge from S, and if a set of vw-paths are edge disjoint then each edge in S is contained in at most one of the paths. It is possible to prove that the the maximum number of edge-disjoint vw-paths is at least equal to the size of the smallest vw-disconnecting set through the use of strong induction on the number of edges in G (although we will not cover the details of this proof here.)

Example 7.19. We saw in Example 7.12 that the edge-connectivity of K_n is at most $n-1$. We can use Menger's theorem to show that this is indeed the correct value. If S is any disconnecting set for K_n, then we can label the vertices of K_n with the labels $v_1, v_2, v_3, \ldots, v_n$ in such a way that v_1 and v_n lie in different components of $K_n - S$. Then S is a v_1v_n-disconnecting set.

The path v_1v_n, together with the paths $v_1v_iv_n$ for $i = 2, 3, \ldots, n-1$ form a set of $n-1$ edge-disjoint v_1v_n-paths. This implies that the minimum size of any v_1v_n-disconnecting set is $n-1$. Hence for any disconnecting set S, we have $|S| \geq n-1$, and thus the edge-connectivity of K_n is precisely $n-1$.

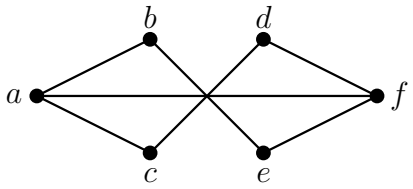
An extension of the reasoning in the above example leads to the following corollary:

Corollary 7.6. *A connected graph G is k-edge connected if and only if for every pair of vertices v and w in G there exists a set of k edge-disjoint vw-paths in G.*

We note that while Menger's theorem guarantees the existence of a set of edge-disjoint vw-paths, it does not specify a means of finding such a set of paths.

We also observe that since Menger's theorem gives the correspondence between the size of the *smallest* vw-disconnecting set and the *largest* possible set of edge-disjoint vw-paths, it follows that if we can find vw-disconnecting set in a graph G that is the same size as some set of edge disjoint vw-paths, we can conclude that the vw-disconnecting set is as small as possible and the set of edge-disjoint vw-paths is as large as possible in G.

Example 7.20.



The set of edges $\{af, be, cd\}$ is an af-disconnecting set for this graph. The paths $acdf$, af and $abef$ are all edge disjoint. Hence we conclude that there is no large set of edge-disjoint af-paths, nor any smaller af-disconnecting set.

There is another form of Menger's theorem that pertains to vertex-disjoint paths and vw-separating sets:

Theorem 7.7 (Menger's Theorem: Vertex Form)**.** *If v and w are two non-adjacent vertices of a connected graph G, then the maximum number of vertex-disjoint vw-paths in G is equal to the size k of the smallest vw-separating set.*

This can be proved in a similar manner to the edge form. As before, we have a corollary:

Corollary 7.8. *A connected graph G with $\geq k+1$ vertices is k-connected if and only if for any two non-adjacent vertices v and w there is a set of k vertex-disjoint vw-paths.*

Exercise 7.21.

1. Determine the edge-connectivity of the Petersen graph.
2. What is $\lambda(K_{2,3})$? What is $\kappa(K_{2,3})$?
3. Give an example of a graph that is 3-edge connected, but whose connectivity is just 1.

7.2.4 Flows in Networks

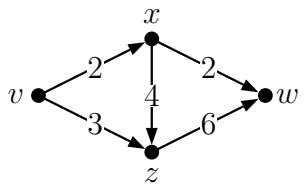
In this section we will see how weighted digraphs can be used in applications such as modelling the flow of water through a system of pipes, or current passing through an electric circuit, *etc.*

Definition 7.11. A *network* (D, ψ) consists of a digraph D, together with a function ψ from the arcs of D to $\mathbb{R}$ that assigns a non-negative real number to each arc of D. The value $\psi(a)$ is referred to as the *capacity* of the arc a.

If the arcs are representing pipes, we can think of the capacity of an arc as the maximum amount of water that can flow along the pipe in a given amount of time.

Definition 7.12. The *indegree* of a vertex v of D is the sum of the capacities of all arcs of D of the form uv. The *outdegree* of v is the sum of the capacities of all arcs of the form vx. A vertex with indegree 0 is known as a *source*. A vertex with outdegree 0 is referred to as a *sink*.

Example 7.22.



In the above network, the capacity of arc vx is 2, the capacity of arc xz is 4, and so on. The indegree of z is 7, the outdegree of z is 6. Node v has indegree 0 and hence it is a source. Node w has outdegree 0 and is thus a sink.

Note that unless otherwise stated, in the rest of this section we consider networks that have precisely one source, and one sink.

Lemma 7.9. *The sum of the indegrees of all the vertices of a network* (D, ψ) *is equal to the sum of the outdegrees of all the vertices.*

Definition 7.13. A *flow* in a network (D, ψ) is a function φ that associates a nonnegative real number $\varphi(a)$ with each arc a of D, such that the following properties hold:

1. $\varphi(a) \leq \psi(a)$ for each arc a of D;
2. if v is any vertex other than the source and the sink then sum of $\varphi(a)$ over all arcs a into v (referred to as the *inflow* of v) is equal to the sum of $\varphi(a)$ over all arcs a out of v (referred to as the *outflow*).

The first of these conditions is akin to saying that the amount of water flowing through a given pipe cannot exceed the capacity of that pipe. The second condition mandates that the amount of water flowing into a junction between pipes is equal to the amount of water that flows out the other side. (In the context of electricity flowing in a circuit, the second of these conditions is sometimes known as Kirchhoff's current law.)

Example 7.23. An example of a flow for the network shown in Example 7.22 is $\varphi(vx) = 2, \varphi(xw) = 1, \varphi(xz) = 1, \varphi(vz) = 3, \varphi(zw) = 4$. (Check for yourself that conditions 1. and 2. of Definition 7.13 are satisfied throughout the network.)

Definition 7.14. A flow φ on a network (D, ψ) for which $\varphi(a) = 0$ for all arcs a is known as the *zero flow* on (D, ψ). An arc a for which $\varphi(a) = \psi(a)$ is said to be *saturated*; an arc a for which $\varphi(a) < \psi(a)$ is *unsaturated*. We observe that the second condition of Definition 7.13, together with Lemma 7.9 implies that the outdegree in (D, φ) of the source is equal to the indegree of the sink; this is known as the *value* of the flow.

Example 7.24. Consider the flow described in Example 7.23. In this flow the arc vx is saturated, but the arc xz is not. The value of this flow is 5.

For many applications we are interested in flows on a network with the maximum possible value; such a flow is known as a *maximum flow*. For instance, we may wish to determine the maximum amount of water that can flow through a network of pipes from a source to a sink in a given amount of time.

Example 7.25. We can see that the flow of Example 7.23 is a maximum flow, as every arc out of the source is saturated. (Note that this is a sufficient condition, but *not* a necessary condition for a flow to be a maximum flow.)

Definition 7.15. Let (D, ψ) be a network with source v and sink w. A *cut* on D is a vw-disconnecting set. The *capacity* of a cut is equal to the sum of the capacities of the arcs in the cut. A cut with the smallest possible capacity is a *minimum cut* for (D, ψ).

Example 7.26. In the network of Example 7.22, one example of a cut is the set $\{vx, zw\}$, as the deletion of these arcs destroys all (directed) paths from the source to the sink. The capacity of this cut is 8. The set $\{vx, vz\}$ is a cut of capacity 5; this is the smallest capacity of any cut for this network.

Just as Menger's theorem allowed us to connect vw-disconnecting sets in a graph with sets of edge-disjoint vw-paths, we have a related theorem for networks that links the value of a maximum flow with the minimum capacity of a cut, the appropriately-named *maximum flow-minimum cut theorem*.

Theorem 7.10 (The Maximum Flow-Minimum Cut Theorem)**.** *Let (D, ψ) be a network with a single source v and a single sink w. The minimum possible capacity of a cut in D is equal to the value of a maximum flow.*

Example 7.27. We have noted that a minimum cut for the network of Example 7.22 has capacity 5. The flow for this network described in Example 7.23 has value 5, and hence is a maximum flow by the maximum flow-minimum cut theorem.

Corollary 7.11. *If (D, ψ) is a network in which all the capacities are integers, then the value of a maximum flow is an integer.*

Learning Outcomes

After completing this chapter and the related problems you should be able to:

- Use the shortest path algorithm to find shortest paths between vertices in a weighted graph or network;

- Display the relationships between tasks that form part of a complex project in the form of an activity network;
- Apply the longest path algorithm to find the critical path in an activity network;
- Describe how to determine the earliest possible start time and latest possible start time of an activity that is represented by an arc of an activity network;
- Be familiar with the concepts of connectivity and edge connectivity of a graph, and their relationship with disconnecting sets, separating sets, cutsets, and vertex cutsets;
- Determine the edge connectivity or the connectivity of a given graph;
- State and apply Menger's theorem in both the edge form and the vertex form;
- Recall the definition of a flow in a network;
- State and apply the maximum flow-minimum cut theorem.