Chapter 6

Graphs

6.1 Introduction

The idea of a graph (not in the figure or plotting sense, but as a structure) was introduced in the 18^{th} century to ask questions about paths. The idea now appears everywhere, but transport maps remain one of the key examples. A graph will abstract away all of the details and only show destinations (usually as points) and connecting paths between destinations (usually as lines), together with minimal extra information (such as the distance along a path, or the cost to travel along it).

The mathematical structure can be used to answer a number of detailed questions, but one key question is fundamental. Given a graph with paths linked to distances, what is the shortest distance between two given points?

6.2 Directed graphs

The mathematical definition of a *(directed) graph* is the ordered pair $G = (V, A)$, where the sets are

- the set of *vertices* (or *nodes*) V, with $|V| = n$;
- a set of ordered pairs $A \subseteq V \times V$, excluding self loops, called *arcs*, with $|A| = m$.

Given an arc $(i, j) \in A$, the node i is called the *tail* and node j the *head*. In a directed graph, travel is only possible from the tail to the head (so imagine drawing an arrow on the arc).

We have not explicitly said what space the nodes are drawn from. In general it does not matter. When giving mathematical examples it is typical to use the natural numbers, so $V = \{1, \ldots, n\}$, and the node number labels the destination. When thinking about concrete transport examples, it can make more sense to use a name or an alphanumeric code. A simple directed graph using integers is illustrated in figure 6.1.

6.2.1 Definitions

We call a node $j \in V$ *adjacent* to another node $i \in V$ if $(i, j) \in A$. As, by construction, there are no self-loops allowed, a node cannot be adjacent to itself.

We call an arc $a \in A$ *incident* to a node $j \in V$ if $a = (i, j)$ for some node $i \in V$. So an arc is incident on a node if there is some other node in the graph that uses the arc to move to the node.

We call a directed graph G *complete* if A contains an arc for each distinct pair of nodes,

$$A = \{(i, j) \colon \forall i \in V, j \in V, i \neq j\}\,. \tag{6.1}$$

If the graph is complete we can immediately travel from any node to any other node using just one arc.

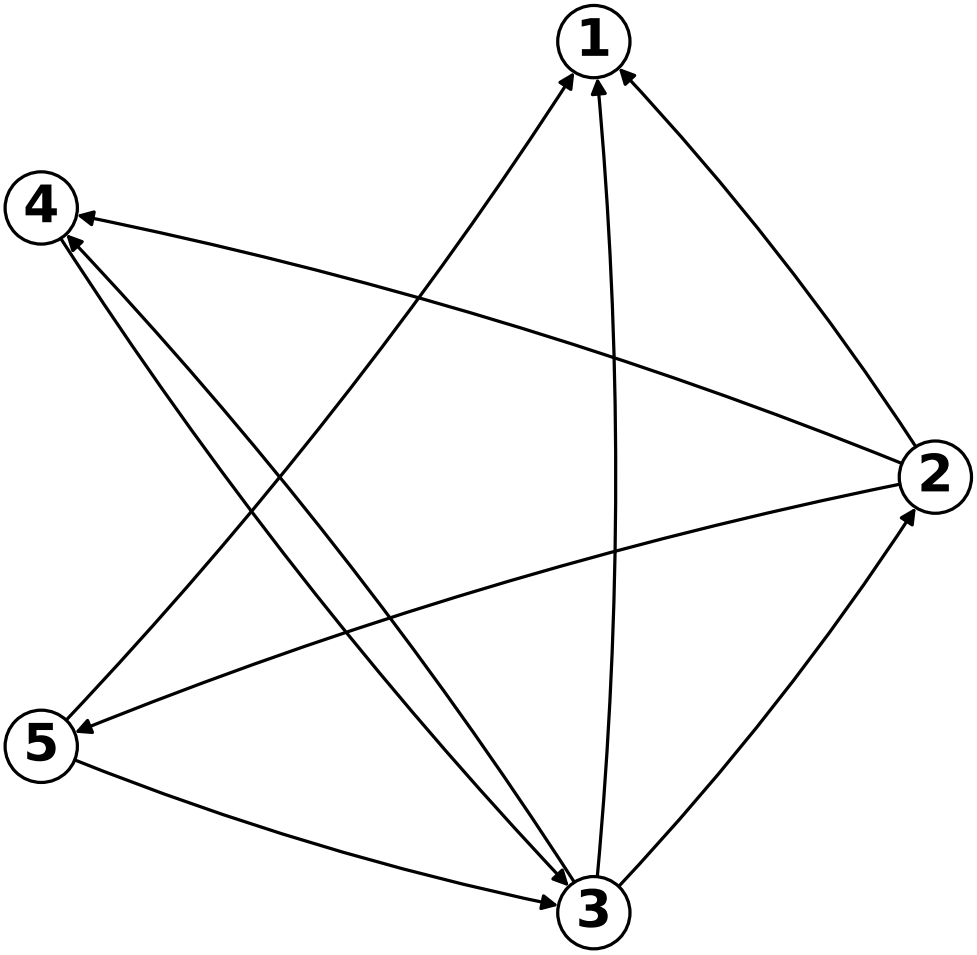

Figure 6.1: A simple directed graph. The vertices or nodes are the circles, here labelled by natural numbers. The edges are illustrated as arrows.

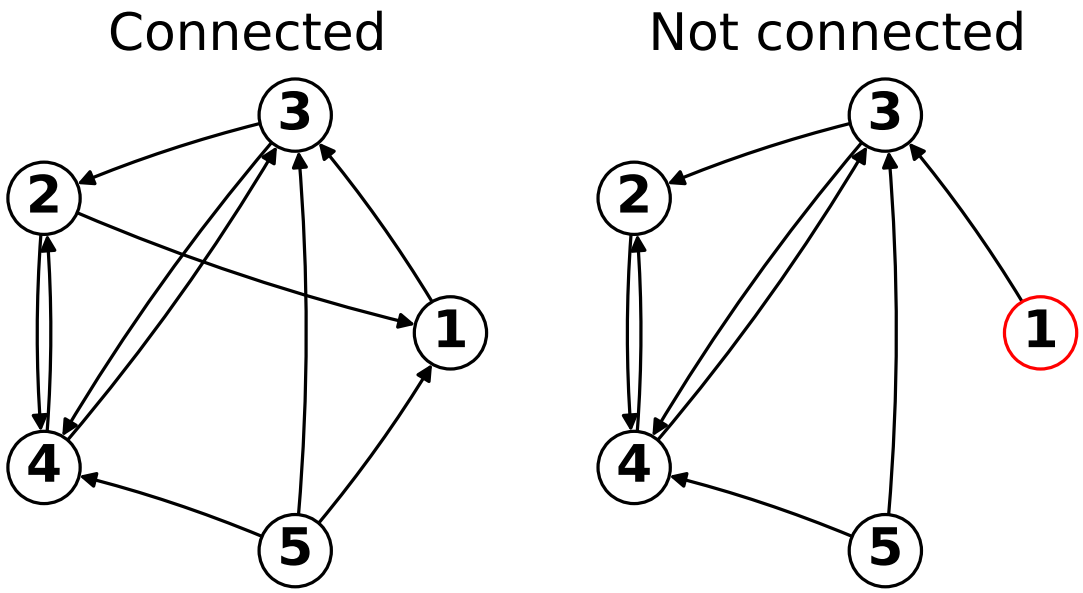

Figure 6.2: The graph on the left is connected but not complete. The graph on the right is not connected, as no edge goes in to node 1.

6.2.2 Number of arcs

Theorem 6.2.1. *In any directed graph, $m \leq n(n-1)$, with $m = n(n-1)$ in the complete case.*

Proof. In the complete case, each of the n nodes is adjacent to all of the other $(n-1)$ nodes: we have $m = n(n-1)$ arcs.

If the graph is not complete, some arcs are missing. Hence $m \leq n(n-1)$. □

We called a directed graph *sparse* if $m \ll n(n-1)$.

6.2.3 Paths

If the graph is not complete then traversing a single arc may not be enough to get us from our start point to the desired end point. Instead we may have to traverse multiple arcs.

We define a *path* as a sequence

$$P = (i_1, i_2), (i_2, i_3), \ldots, (i_{k-1}, i_k), (i_k, i_{k+1}) \tag{6.2}$$

of $k \geq 1$ consecutive and distinct arcs. Note that the head of each arc in the sequence matches the tail of the next arc, making the arcs consecutive.

Given a path, we say that a node $v \in V$ is *connected* to a node $w \in V$ if there is a path P with $i_1 = v$ and $i_{k+1} = w$.

Finally, we say that a graph is *connected* if every pair of its nodes is connected. This is illustrated in figure 6.2.

6.2.4 Cuts

So far everything has been set either at the level of individual nodes or at the level of the whole graph. For building algorithms that allow us to analyse paths as they move through the graph, we need to look at subsets of the graph, and how we might move into or out of those subsets.

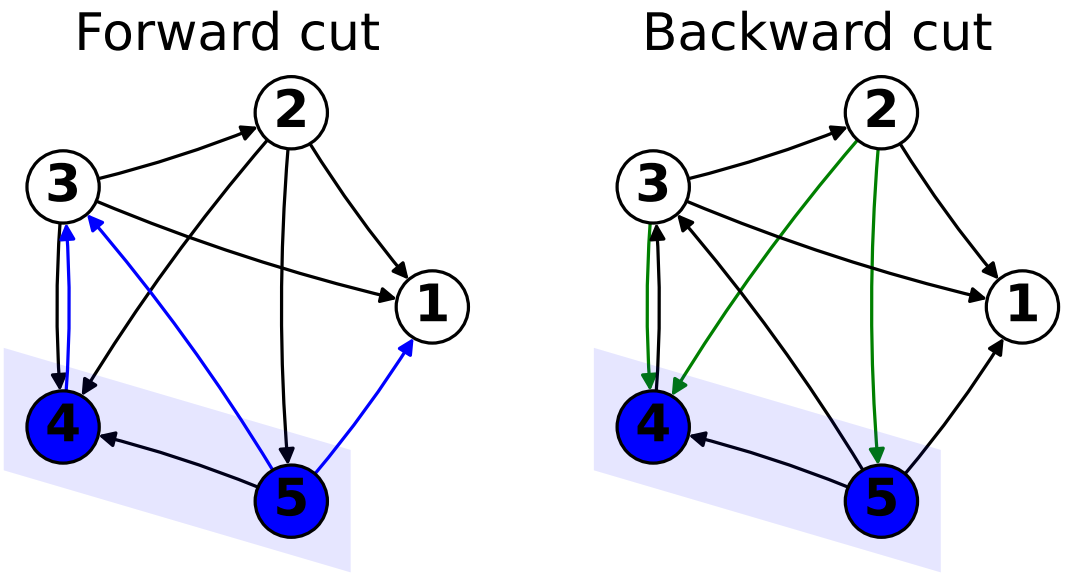

Figure 6.3: The set $S = \{4, 5\}$ is shaded in blue. Its forward cut $\delta^+(S)$ is every edge (highlighted in blue in the left plot) that leaves S. Its backward cut $\delta^-(S)$ is every edge (highlighted in green in the right plot) that enters S.

Let $S \subseteq V$ be some part of graph (by looking at some suset of the vertices). We define the *forward cut induced by* S to be the set of arcs "leaving S". That is

$$\delta^+(S) = \{(i, j) \in A \colon i \in S \text{ and } j \in V \setminus S\}\,. \tag{6.3}$$

Similarly, we define the *backward cut induced by* S to be the set of arcs "entering S". That is

$$\delta^+(S) = \{(i, j) \in A \colon i \in V \setminus S \text{ and } j \in S\}\,. \tag{6.4}$$

These are illustrated in figure 6.3.

Of course, we can apply these definitions to the case to single nodes, by setting $S = \{i\}$ for any node $i \in V$. These have special names: the *forward star* of the node i is $\delta^+(i)$, whilst the *backward star* of the node i is $\delta^-(i)$. The sizes of these sets also have special names, with the *out-degree* of node i being $|\delta^+(i)|$, and the *in-degree* of node i being $|\delta^-(i)|$.

6.2.5 Representations

So far we have represented the graph using the vertices V and the arcs A. However, as in the linear programming case, there are multiple ways of representing the problem, each of which has its own advantages.

For simplicity these additional representations will assume that V is given by consecutive integers labelling the nodes. Depending on choice these can either start from 1 (so $V = \{1, \ldots, n\}$) or 0 (so $V = \{0, \ldots, n-1\}$). The latter is more natural for a Python implementation, but the former more natural in many mathematical texts. With this assumption we only need to consider different representations of the arcs, A.

Adjacency list

In the *adjacency list* approach we construct a list L of size n. Each component of the list L_i contains a list of size at most $n-1$, containing the indices of the nodes adjacent to i. In terms of

the stars, we have

$$L_i = \{j \colon (i,j) \in \delta^*(i)\}. \tag{6.5}$$

The advantage of the adjacency list approach is that it is easy to use to navigate the graph. The component L_i gives every node that is reachable from node i using a single arc.

Adjacency matrix

In the *adjacency matrix* approach we construct a single matrix of size $n \times n$ that contains only zeros and ones. The matrix entry is one if the associated arc exists and zero otherwise. Explicitly,

$$m_{ij} = \begin{cases} 1 & \text{if } (i,j) \in A \\ 0 & \text{otherwise} \end{cases}. \tag{6.6}$$

The adjacency matrix is less efficient for navigating the graph, but more efficient if we need to check if an arc exists.

6.3 Shortest paths

Now that we have the terminology to represent and discuss graphs, we want to turn the word problem "Find the shortest path connecting two points" into something precise.

6.3.1 The problem

The *shortest path problem* is as follows. Given a directed graph $G = (V, A)$ with a (non-negative) length function $l \colon A \to \mathbb{R}^+$ and two nodes $s, t \in V$, find an $s - t$ path of shortest total length.

The length function tells us the distance along any one single arc: equivalently, the distance between any two nodes or vertices when moving in a specific direction. This need not be the same in both directions (hence the notion of a *directed* graph): think about one-way streets, for example. As a notational shortcut we will often talk about lengths of paths in addition to lengths of arcs. As a path is a sequence of consecutive arcs without loops and the length function is non-negative, it follows that

$$l(P) = \sum_{a \in P} l(a). \tag{6.7}$$

If the arc $a = (i,j)$ then sometimes the notation $l_{ij} = l((i,j)) = l(a)$ is used, and so we can write

$$l(P) = \sum_{(i,j) \in P} l_{ij}. \tag{6.8}$$

The two nodes in the problem are the *source* node s from which we start and the *target* node t that we are trying to get to.

There is a generalisation of the problem which is, in fact, no harder to solve. The *single source shortest path problem* is as follows. Given a directed graph $G = (V, A)$ with a (non-negative) length function $l \colon A \to \mathbb{R}^+$ and one node $s \in V$, find a path between s and every other node in $V \setminus \{s\}$ of shortest total length.

We note that it only makes sense to solve the shortest path problem if the source and target nodes are connected. Therefore it only makes sense to solve the single source shortest path problem if the entire graph is connected. Throughout the rest of this chapter we will assume that the graph is connected.

6.3.2 Subpath optimality

The key result in finding the shortest path is intuitively stated as "shortest paths are composed of shortest paths". This seems either obvious or trivial. The precise result is

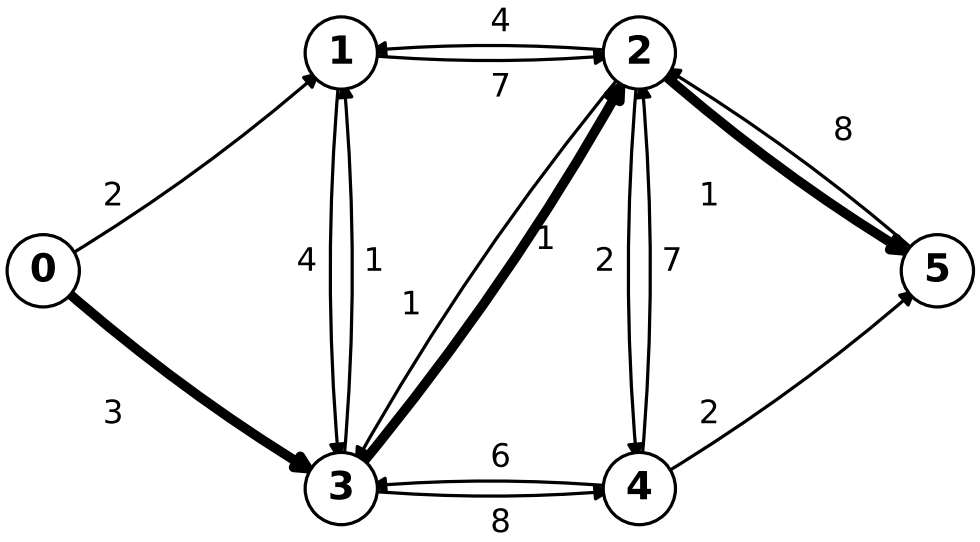

Figure 6.4: A graph with its shortest path. Each node is annotated with its label (a natural number here). Each edge (i, j) is annotated with its length l_{ij}. The bold lines show the shortest $0-5$ path, $P = (0,3),(3,2),(2,5)$.

Theorem 6.3.1 (Subpath Optimality)**.** *Let $P = (i_1, i_2), (i_2, i_3), \ldots, (i_{k-1}, i_k)$ be an $i_1 - i_k$ shortest path. For any pair of nodes i_u, i_v visited by P, with $u < v$, the subpath S from i_u to i_v is a shortest $i_u - i_v$ path.*

Proof. This follows by contradiction. if S is not a shortest $i_u - i_v$ path then there is another $i_u - i_v$ path S' that is strictly shorter than S, $l(S') < l(S)$. We can therefore define a new $i_1 - i_k$ path P' by using the "shortcut" S',

$$P' = P \setminus S \cup S'. \tag{6.9}$$

From the positivity of the length function it immediately follows that

$$l(P') = l(P) - l(S) + l(S') < l(P). \tag{6.10}$$

Therefore P was not the shortest path, and we have a contradiction. □

The original characterisation can now be more precisely stated as "every subpath of a shortest path is itself a shortest path".

6.4 Dijkstra's theorem

This is the central result that allow us to build an algorithm to solve the shortest path problem.

Theorem 6.4.1 (Dijkstra's theorem)**.** *Let $S \subseteq V$ be a subset of the vertices that contains the source s. Let $Y(i)$, for all $i \in V$, be the length of the corresponding shortest $s - i$ path. Let*

$$(v, w) \in \underset{(i,j)\in\delta^+(S)}{\operatorname{argmin}} \left[Y(i) + l_{ij}\right]. \tag{6.11}$$

Then $\varphi = P_{(s-v)} \cup \{(v, w)\}$ is a shortest $s - w$ path, where $P_{(s-v)}$ is a shortest $s - v$ path.

Proof. We show that any other path π to any other vertex $u \in V \setminus S$ has either the same length as φ or a strictly larger one.

First, decompose the path π as

$$\pi = \pi_1 \cup \{(i,j)\} \cup \pi_2, \tag{6.12}$$

where $i \in S$, $j \in V \setminus S$, π_1 is a shortest $s - i$ path, $(i,j) \in \delta^+(S)$, and π_2 is a shortest $j - u$ path. This is always possible due to the subpath optimality theorem. It allows us to concentrate on the arc (i,j) that leaves the set S. It follows that

$$l(\pi) = l(\pi_1) + l_{ij} + l(\pi_2). \tag{6.13}$$

Since we have chosen $(v,w) \in \delta^+(S)$ to minimise $Y(i) + l_{ij}$ it must be true that

$$l(\pi_1) + l_{ij} \geq Y(v) + l_{vw}. \tag{6.14}$$

Since $l(\pi_2) \geq 0$ it follows that

$$\begin{aligned} l(\pi) &= l(\pi_1) + l_{ij} + l(\pi_2) && (6.15)\\ &\geq l(\pi_1) + l_{ij} && (6.16)\\ &\geq Y(v) + l_{vw} && (6.17)\\ &= l(\varphi). && (6.18) \end{aligned}$$

Therefore $l(\pi) \geq l(\varphi)$ for all such paths, and therefore φ is a shortest $s - w$ path. □

6.4.1 Dijkstra's algorithm

We can now use subpath optimality and iteratively apply Dijkstra's theorem to give us an algorithm. Starting from the source node, we keep adding nodes to the set of nodes we have "seen" so that we have the shortest path in that set.

Being more precise, but still sketching in words. We want to find the shortest path from the source node s to every other node in the graph. We construct a set $S \subseteq V$, starting from $S = \{s\}$, of all the nodes we have so far "seen". We also need two *labels*. The first is $Y(i)$, the shortest path length from s to i. The second is $P(i)$, the predecessor of i in the $s - i$ path. At the start $Y(s) = 0$ and $P(s)$ is undefined (as it makes no sense).

We then iterate as long as $|S| < n$, so that $V \setminus S$ is not empty. For each iteration we

1. find $(v,w) \in \delta^+(s)$ that minimises $Y(v) + l_{vw}$;
2. set $Y(w) = Y(v) + l_{vw}$;
3. set $P(w) = v$;
4. set $S \to S \cup \{w\}$.

Note that, for every node i, the shortest path length from the source $Y(i)$ adn the predecessor node on the shortest path $P(i)$ are set only once, when the final shortest arc to i is found and i is added to S.

This solves the *single source* shortest path problem. To solve the shortest path problem when the target t is specified we can check to see if $w = t$ and, if so, stop at the end of that iteration. This makes no difference to the code complexity as, in the worst case that has to be considered, t will be the last node added to S.

6.4.2 Representing the length

We have seen multiple methods for representing the topology of the graph, each of which has their individual advantages. As Dijkstra's algorithm depends crucially on minimising a length and looking up l_{vw}, it is important to have an easy and efficient way of representing the arc lengths.

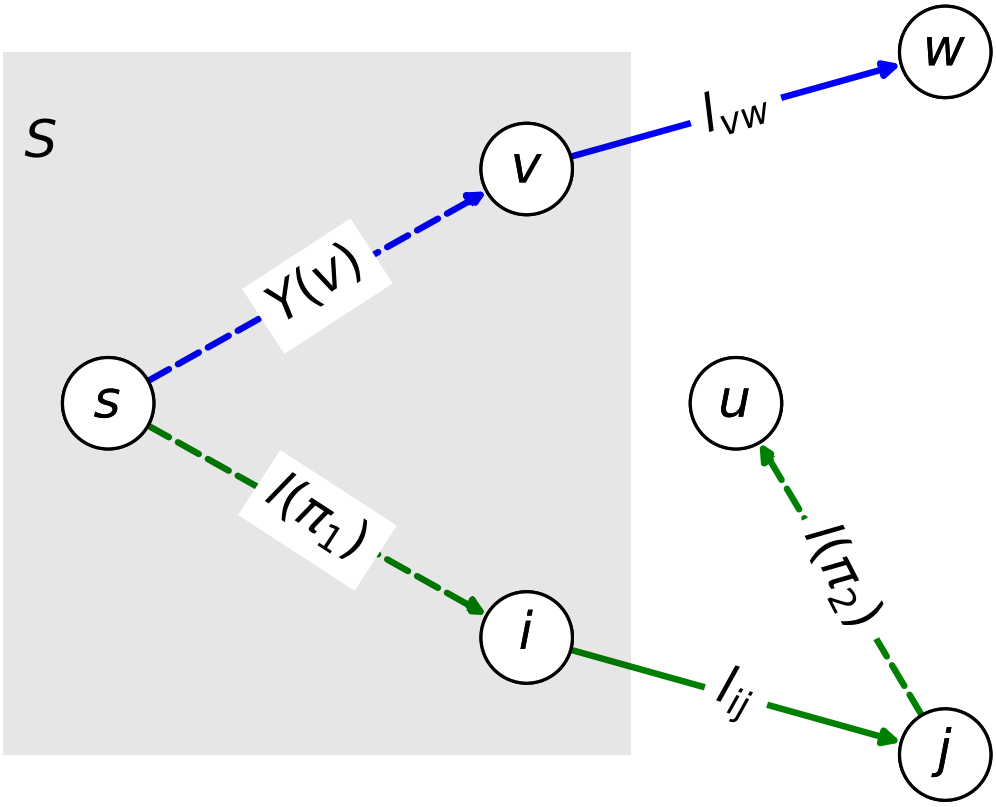

Figure 6.5: The key idea used in proving Dijkstra's theorem. We want to show that the shortest path from s to w is given by the shortest $s-v$ path (already found an in S, by assumption) linked to the $v-w$ edge. This must be true if the length of this path is no greater than the length of any other path connecting any point not in S to s.

Adjacency list

As in the graph case, we construct a list L_l of size n, where each component i contains another list of size at most $n-1$, containing the *lengths* of all arcs $(i,j) \in \delta^+(i)$, in order of index j.

This representation needs the adjacency list L as well in order to give the index j. Assuming that V is a set of consecutive integers of size n, the whole graph (including arc length information) is given once L, L_l are given. The size of the graph n is implicit in the length of the lists.

Adjacency matrix

Again this is similar to the graph case. We construct a matrix $M_l \in \{\mathbb{R} \cup \{\infty\}\}^{n \times n}$ where, for each $i, j \in V$, the component $(m_l)_{ij}$ is the length of the (i,j) arc, or is infinity if no such arc exists.

This representation implicitly stores all the information about the graph, including its topology, so the adjacency matrix M is not needed.

6.4.3 Implementing Dijkstra's algorithm

The adjacency list form is the most efficient for checking whether an arc exists. The adjacency matrix form is the most efficient for checking what the length of the arc is. We assume that we are not limited by the amount of memory that each form takes, so we will use both forms in our implementation. In the code snippet below, `L` is the adjacency list representing the topology of the graph and `Ml` is the adjacency matrix representation of the length function.

```
1  S = [s]
2  Y = np.empty(n)  # Value does not matter
3  P = np.empty(n)  # Only size matters
4  Y[s] = 0
5  while len(S) < n:
6      min_Y = np.inf
7      v, w = -1, -1
8      for i in S:
9          if j in S:  # Only consider arcs leaving S
10             continue
11         if min_Y > Y[i] + Ml[i][j]:
12             min_Y = Y[i] + Ml[i][j]
13             v, w = i, j
14     Y[w] = Y[v] + Ml[v][w]
15     P[w] = v
16     S.append(w)
```

6.4.4 Complexity

To compute the complexity we consider the order of each line or block of the code above. We have

- Line 1 writes a single value, taking $\mathcal{O}(1)$;
- Lines 2-3 write n values, taking $\mathcal{O}(n)$ each;
- Line 4 writes a single value, taking $\mathcal{O}(1)$;
- Line 5 loops over the whole graph, so is executed n times. Each block takes:
  - Lines 6-7 write single values, taking $\mathcal{O}(1)$;
  - Lines 8-10 correspond to looping over $\delta^+(S)$ in its entirety. This is executed m times, where m is the out degree of S. Earlier results show $m \leq n(n-1)$. In the worst case $m = \mathcal{O}(n^2)$. Each block takes:

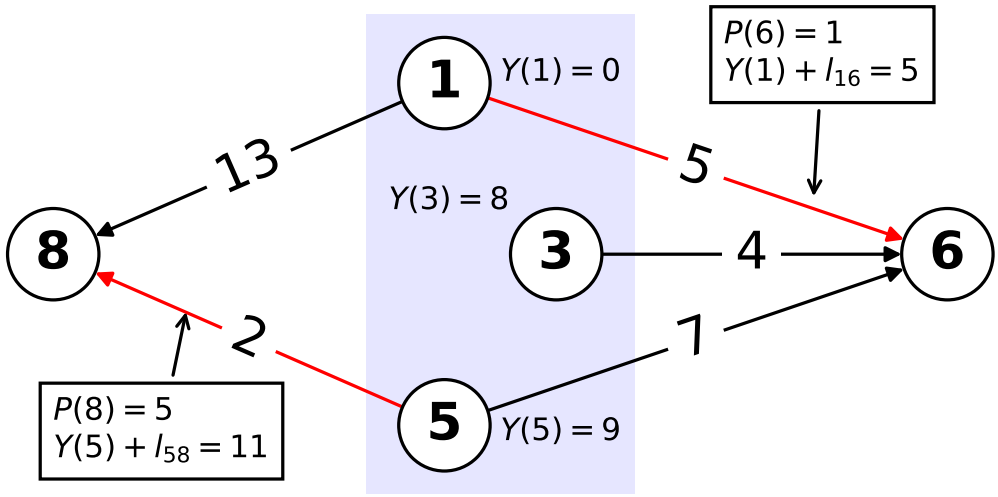

Figure 6.6: The key idea used in constructing the faster $\mathcal{O}(n^2)$ version of Dijkstra's algorithm. When trying to add to the set S (blue shaded region) we must compute the shortest path and predecessors to the points not in S, given the current S (red edges). By storing these values and updating them when needed, instead of re-computing every iteration, we save lots of computation.

- $*$ Lines 11-13 are single comparisons or single writes, each taking $\mathcal{O}(1)$;

  Therefore this inner loop has complexity $\mathcal{O}(m) = \mathcal{O}(n^2)$.
- Lines 14-16 are single writes, each taking $\mathcal{O}(1)$;

Therefore this outer loop has complexity $\mathcal{O}(n)\mathcal{O}(n^2) = \mathcal{O}(n^3)$.

Therefore the whole algorithm has complexity $\mathcal{O}(n^3)$.

This is considerably faster than the complexity of the simplex algorithm (in its worst case).

6.4.5 A faster method

Finding the shortest path is a sufficiently important practical problem that we want to find an algorithm faster than $\mathcal{O}(n^3)$. This can be done. The key issue is that by repeatedly scanning $\delta^*(S)$ we are recomputing particular shortest path lengths multiple times. With additional structures we can reduce that computation.

We need the following observation:

$$(v, w) \in \operatorname*{argmin}_{(i,j)\in\delta^+(S)} \{Y(i) + l_{ij}\} \tag{6.19}$$

$$\iff \quad (v, w) \in \operatorname*{argmin}_{j\in V\setminus S} \left\{ \operatorname*{argmin}_{(i,j)\in\delta^-(S):\ i\in S} \{Y(i) + l_{ij}\} \right\}. \tag{6.20}$$

The point here is that if we already know, for given node $j \in V \setminus S$, which arc minimises $Y(i)+l_{ij}$, then we can find the arc (v, w) giving the shortest path by looking at the $\mathcal{O}(n)$ vertices $j \in V \setminus S$, rather than looking at the $\mathcal{O}(m) = \mathcal{O}(n^2)$ arcs in $\delta^+(S)$.

To use this observation to construct an algorithm, we change the way we think about the labels $Y(j), P(j)$, representing the length of the shortest $s - j$ path and the predecessor index in that shortest path respectively. In the original algorithm they were set *only when* j enters S. Until that point they had no value, or their value was meaningless.

In the new algorithm, the meaning of the labels will remain the same *as long as* $j \in S$. If $j \in V \setminus S$ then the value of $Y(j)$ and $P(j)$ will represent the length and predecessor index of the shortest $s - j$ path found *up to the current iteration*. At each iteration we will check *and update* the value of the labels when a new vertex enters S.

```
1  S = [s]
2  Y = np.copy(Ml[s][:])   # These values now matter
3  P = np.zeros(n)         # These also now matter
4  Y[s] = 0
5  while len(S) < n:
6      min_Y = np.inf
7      w = -1              # Note: only w, not v
8      for j in V:         # Note: change here
9          if j in S:  # Only consider arcs leaving S
10             continue
11         if min_Y > Y[j]:
12             min_Y = Y[j]
13             w = j
14     S.append(w)
15     for h in L[w]:  # New step: update labels
16         if h in S:
17             continue
18         if Y[h] > Y[w] + Ml[w][h]:
19             Y[h] = Y[w] + Ml[w][h]
20             P[h] = w
```

The explicit complexity calculation is

- Line 1 writes a single value, taking $\mathcal{O}(1)$;
- Lines 2-3 write n values, taking $\mathcal{O}(n)$ each;
- Line 4 writes a single value, taking $\mathcal{O}(1)$;
- Line 5 loops over the whole graph, so is executed n times. Each block takes:
  - Lines 6-7 write single values, taking $\mathcal{O}(1)$;
  - Lines 8-10 correspond to looping over $V \setminus S$ in its entirety. In the worst case this is $\mathcal{O}(n)$. Each block takes:
    - Lines 11-13 are single comparisons or single writes, each taking $\mathcal{O}(1)$;

    Therefore this inner loop has complexity $\mathcal{O}(n)$.
  - Lines 14 is a single write taking $\mathcal{O}(1)$;
  - Lines 15-17 correspond to looping over $V \setminus S$ in its entirety again. In the worst case this is $\mathcal{O}(n)$. Each block takes:
    - Lines 18-20 are single comparisons or single writes, each taking $\mathcal{O}(1)$;

    Therefore this inner loop has complexity $\mathcal{O}(n)$.

  Therefore this outer loop has complexity $\mathcal{O}(n)\mathcal{O}(n) = \mathcal{O}(n^2)$.

Therefore this algorithm has complexity $\mathcal{O}(n^2)$. When the graph is large this can be a substantial improvement in efficiency.

Note that this is not the fastest algorithm that exists. It is possible to construct algorithms with complexities $\sim \mathcal{O}(n \log(n))$.