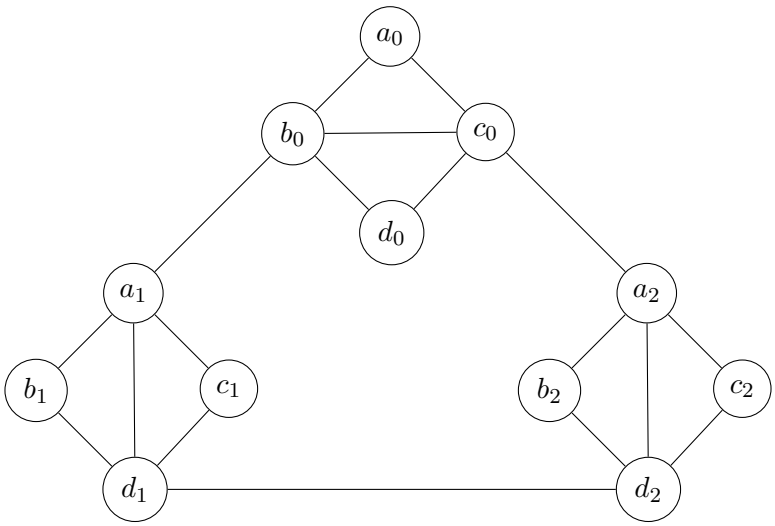

Figure 1: A graph that usefully illustrates Fleury's algorithm.

12- graphs two[1] - DRAFT

In these notes we will learn about two constructive algorithms; the first, Fleury's Algorithm, allows an Eulerian path to be constructed if one exists and the second, Dijkstra's Algorithm, a very famous and useful algorithm, allows us to find the shortest path through a weighted graph. We won't look at a formal proof in either case, but for both the algorithms are compelling enough that it is reasonably clear how a proof might work.

Fleury's algorithm

Imagine trying to find an Eulerian cycle for the graph in Fig. 1, but without thinking about it too hard; say you start at a_0 and go to b_0 before heading off for a_1, then b_1 and d_1 before going hog-wild and going on to d_2, the situation you find yourself in is illustrated in Fig. 2; it is clear the situation is hopeless, there is no way to return to the subscript-1 group of nodes to traverse the edges between a_1 and d_1, a_1 and c_1 and d_1 and c_1. These should have been finished up before leaving for the subscript-2 nodes. The sequence $a_0b_0a_1d_1a_1c_1d_1d_2$, but even if the subscript-2 nodes are all carefully visited, $d_2b_2a_2c_2d_2a_2$ before returning to the subscript-0 nodes, a_2c_0 it is still possible to mess up by returning to a_0 before visiting the edges $c_0d_0b_0c_0$.

Playing with this example shows that the problem is with creating 'islands', a node, or group of nodes, you can't return to because all the links to them have been used up. At this point it is useful to define a *connected* graph and a *bridge*. A graph is connected if, for every pair of nodes a and b there is a trail from a to b. Now, a bridge is an edge which, if it was removed, would leave the graph disconnected; examples are given in Fig. 3. Now imagine removing the edges as you traverse them, if you remove a bridge then you cannot return to the nodes left behind, so when trying to walk an Eulerian path it is important not to remove bridges if there are still edges to walk. This is basically Fleury's algorithm, it says that you should never choose a bridge if there is a choice, this is summarized in Table 1.

[1] `http://github.com/conorhoughton/COMS10001`

- If a node has an edge that isn't a bridge take that.
- As you traverse each edge remove it.
- If you remove the last edge at a node, remove the node.

Table 1: Fleury's algorithm

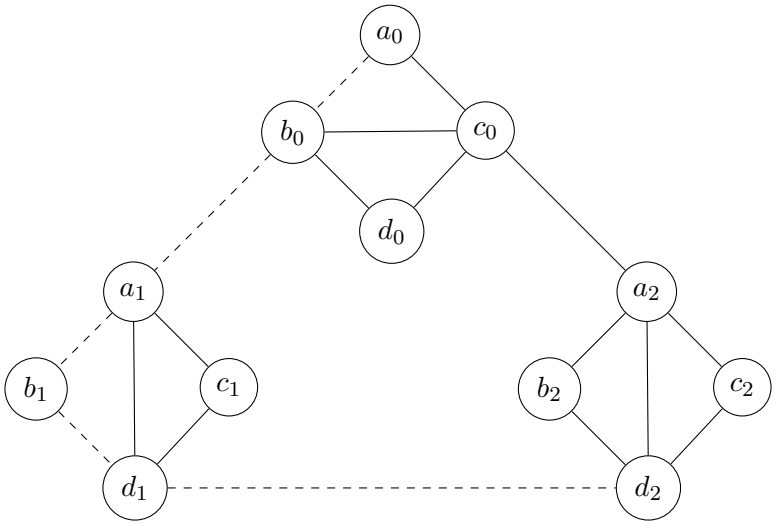

Figure 2: A bad start to finding an Eulerian path, the edges already traversed are dashed.

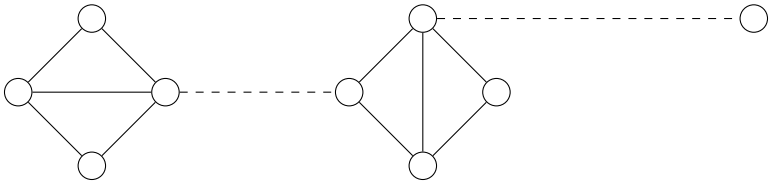

Figure 3: The two dashed edges are bridges since deleting either of them would make the graph disconnected.

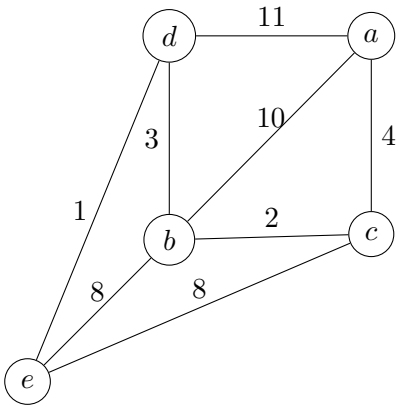

Figure 4: A weighted graph

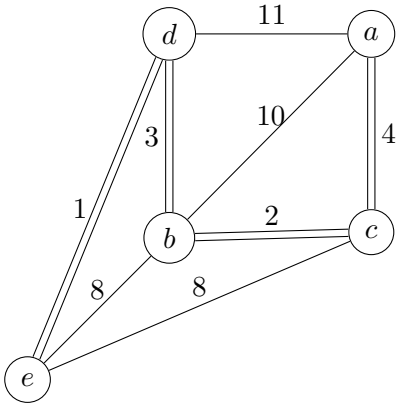

Figure 5: Shortest path: the double lines mark the shortest path

Dijkstra's algorithm

The problem Dijkstra's algorithm solves is how to find the shortest path through a weighted graph. Consider the graph in Fig. 4; Dijksktra's algorithm solves, for example, the problem of how to go from (a) to (e) by the shortest route. The actual shortest route is given in Fig. 5.

Now if you start at (a) you can work out directly the distance using one edge to all the nodes adjacent to a, this is given in Fig. 6. Let's attribute a distance of $D(x)$ to a node x and call the length of the edge between nodes x and y $D(x, y)$. Thus, after looking at the nodes adjacent to a, we have $D(b) = 10$, that is the distance ten is attributed to (b). This is not the actual shortest distance from a to b, there is a shorter path that goes by way of c which would give a distance of six. Thus, the next step is to select another node, let's call it x and update the distances to its neighbours, so, for example, if y is a neighbour of x and $D(x)+D(x, y) < D(y)$ then $D(y)$ should be updated to this new, shorted, distance, $D(y) \to D(x) + D(x, y)$.

Now, this process might reduce some distances, but will never reduce the distance to the node with the lowest $D(x)$; thus, if we choose x to be this lowest distance node, after we update all its connected nodes, we can cross it off and not look at it again. In the case of the graph we

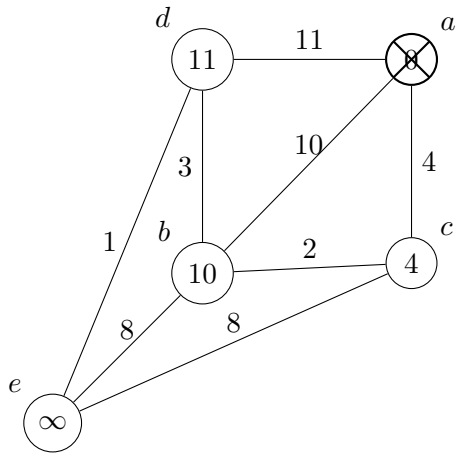

Figure 6: This shows the distance to the nodes adjacent to (a) directly along one edge, the node labels have been moved outside the nodes so that the distances can be written inside. The distance to (e) has been marked as ∞ because we haven't found a route to (e) yet. (a) has a cross in it since we have already looked at its nearest neighbours.

have been looking at, the lowest node in Fig. 6 is c, so the next step is to look at its neighbours, a has already been crossed off, so $D(b)$ is updated to six and $D(e)$ to 12. This situation is shown in Fig. 7. At this point b has the smallest assigned distance, so it is considered next, since $D(b) + D(b, e) = 14 > 12$, $D(e)$ isn't changed, but $D(d)$ is set to nine. This is shown in Fig. 8**A**. Now the smallest node is d, updating its neighbours reduces the distance to e to 10, see Fig. 8**B**. Since e is now the unchecked node with the smallest distance we are done.

It would have been easy to keep track during the algorithm of the node preceding a given node in the shortest path; this would allow the path to be retraced afterwards. This is shown in Fig. 9. A function to implement Dijkstra's algorithm and store the path, can be seen at Table 2.

It is interesting that Dijkstra's algorithm does not use the identity of the target node. Starting with the starting node it spreads knowledge of the shortest path outwards through the graph and terminates when the target node is the available node with the shortest assigned distance.

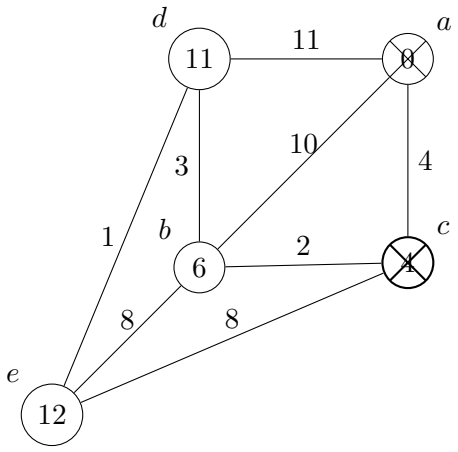

Figure 7: This shows the situation after c's neighbours have been considered.

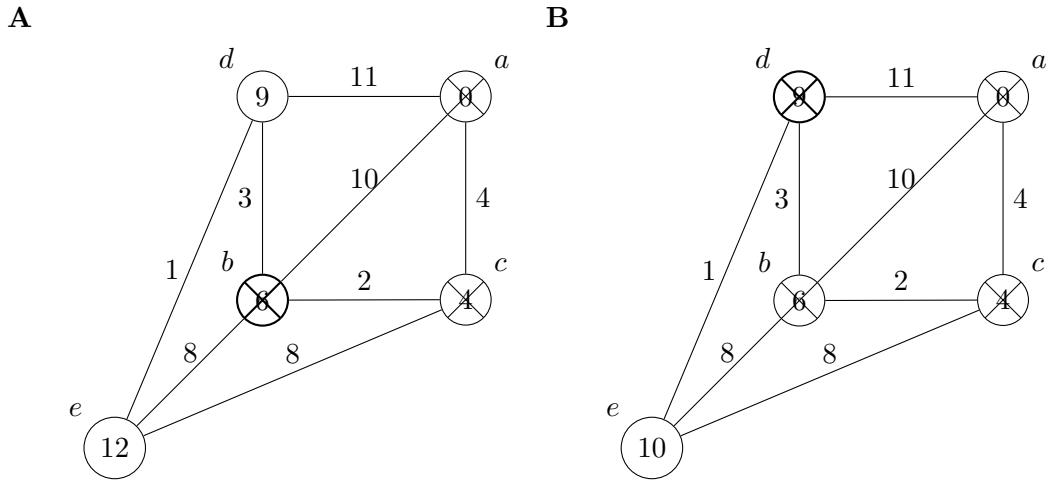

Figure 8: This shows the situation after (**A**) b's and (**B**) d's neighbours have been considered. At this point e, the target node, is the available node with the lowest assigned distance, so the algorithm is done and the shortest path is ten.

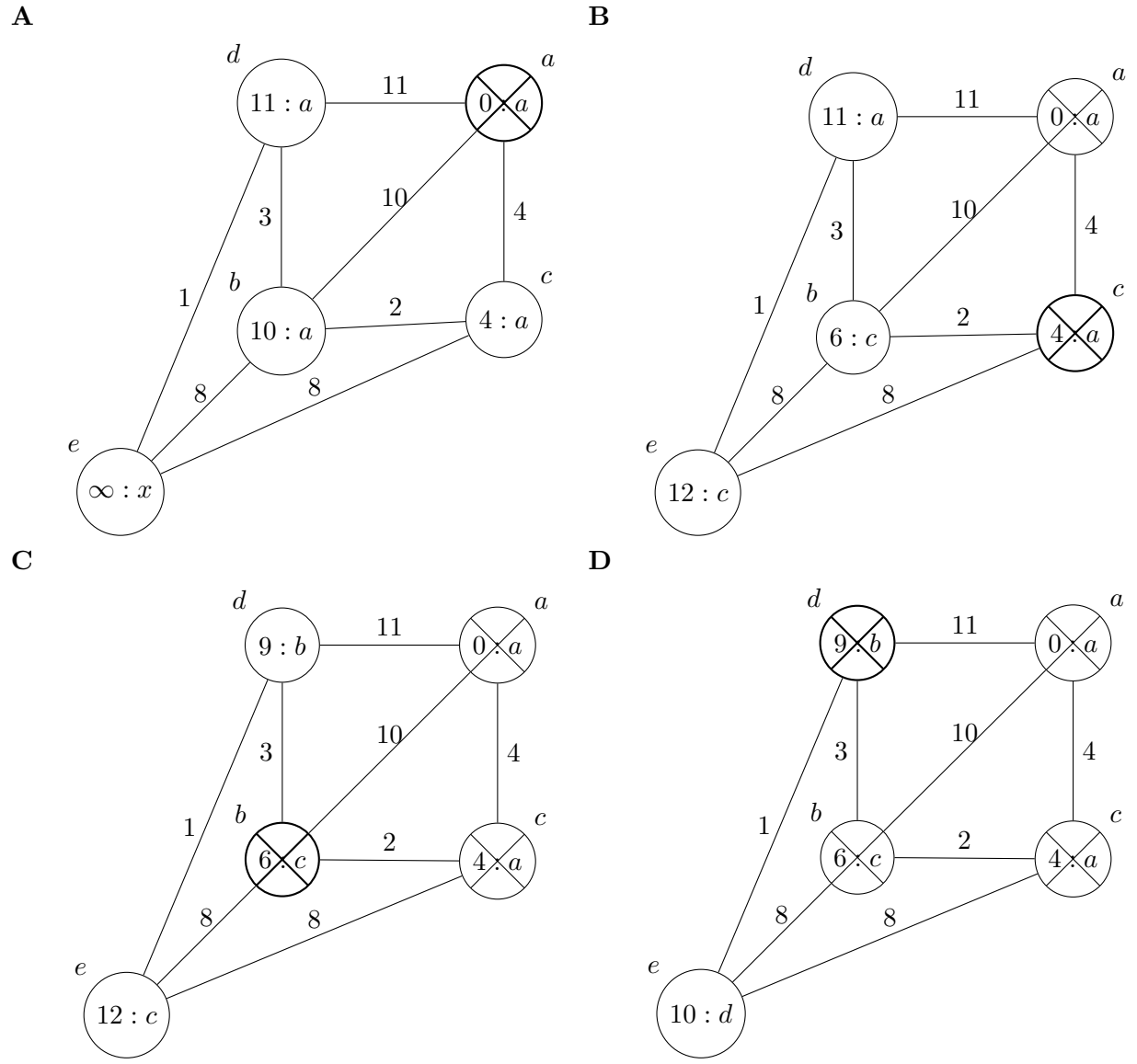

Figure 9: Here along with the assigned distance the previous node is tracked, if the assigned distance is updated, so is the previous node; x is used if there is no previous node, and a is given a as the previous node because it is the start of the path. Running backwards along the nodes we see the shortest path is *edbca*.

```
1  int dijkstra(int a[n][n],int route[n])
2  {
3    int i, available[n], distances[n],current=0,next,min_distance;
4
5    for(i=0;i<n;i++) {
6      route[n]=0;
7      available[i]=1;
8      distances[i]=inf;
9    }
10
11   distances[0]=0;
12
13   while(current!=n-1){
14     available[current]=0;
15
16     for(i=1;i<n;i++){
17       if(available[i]&& distances[i]>distances[current]+a[i][current]){
18         distances[i]=distances[current]+a[i][current];
19         route[i]=current;
20       }
21     }
22
23     next=current;
24     min_distance=inf;
25     for(i=1;i<n;i++){
26       if(available[i]&&distances[i]<min_distance){
27         next=i;
28         min_distance=distances[i];
29       }
30     }
31     current=next;
32   }
33   return distances[n-1];
34 }
```

Table 2: This works out the minimum distance from the 0th node to the `n-1`th node using the distance matrix `a`; it assumes `n`, the number of nodes, is a global variable. This function can been seen in action in `dijkstra.c`.