Graph Contraction

Graph contraction itself is trivial, but what is needed is a method of contracting the graph while retaining all information about the original graph to enable routing points to be re-inserted. Figure 1 illustrates a non-contracted graph, represented by the following structures

```
vert(A).in  = [5]
vert(A).out = [2]
vert(B).in  = [2, 6]
vert(B).out = [3, 5]
vert(C).in  = [3, 7]
vert(C).out = [6, 4]
vert(D).in  = [4]
vert(D).out = [7]
edge(2).verts = [A, B]
edge(3).verts = [B, C]
edge(4).verts = [C, D]
edge(5).verts = [B, A]
edge(6).verts = [C, B]
edge(7).verts = [D, C]
```

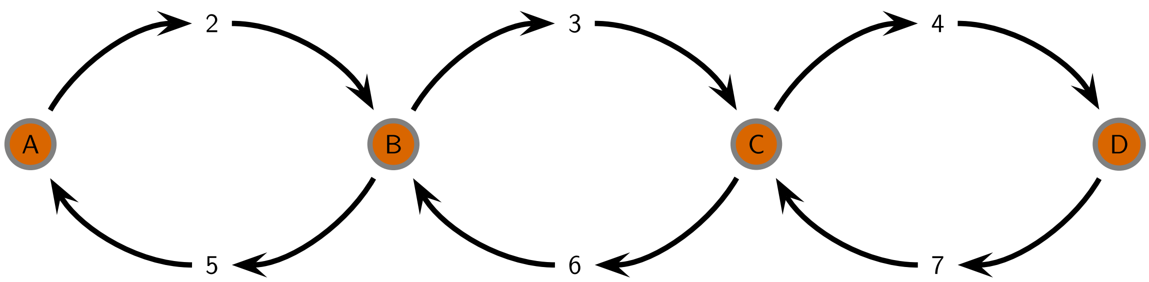

Figure 1: Sample network to be condensed through removal of redundant nodes.

Initial Contraction

Contraction requires a 'vert2edge' map, and a list in each edge of mappings on to original edges in the full graph. The former of these is an unsorted set, initiated as,

```
vert2edge(A) = [2, 5]
vert2edge(B) = [2, 5, 3, 6]
vert2edge(C) = [3, 6, 7, 4]
vert2edge(D) = [7, 4]
```

These could be stored with the vertices themselves, however those vertices have to be deleted following contraction, and this would require construction of a separate list of deleted vertices anyway, in order to allow subsequent re-insertion. Consider a process of contraction which starts by removal of the vertex `B`, with new edges `(2, 3) -> 8` and `(6, 5) -> 9`. These new edges then hold

```
edge(8).old = [2, 3]
edge(9).old = [6, 5]
```

And the 'vert2edge' maps for `A`, `B`, and `C` need to be updated to,

```
vert2edge(A) = [8, 9]
vert2edge(B) = [8, 9]
vert2edge(C) = [8, 9, 7, 4]
```

where `vert2edge(B)` is retained in updated form to allow subsequent re-insertion. The `vert` sets also need to be updated to remove vert B and set

```
vert(A).in  = [9]
vert(A).out = [8]
vert(C).in  = [7, 8]
vert(C).out = [4, 9]
vert(D).in  = [4]
vert(D).out = [7]
```

This reduces the graph to Fig. 2, where the edges `(2, 3, 5, 6)` have simply been removed.

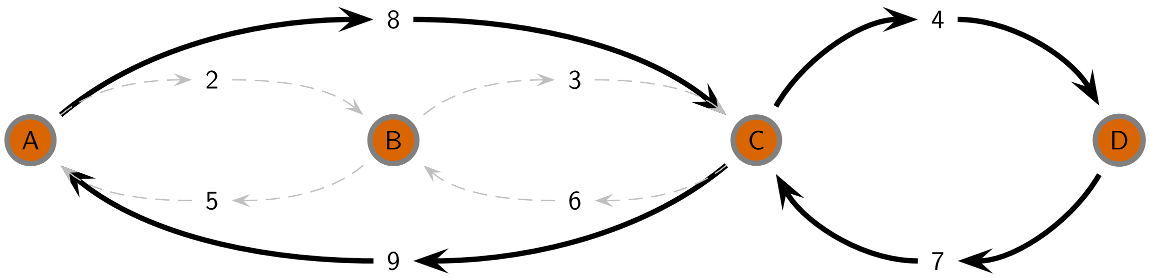

Figure 2: Network of Fig. 1 contracted through removal of vertex B.

The `edge(.).old` sets are *ordered* and so must be constructed with the following pseudo-code:

```
vert2rm = B
# replace vertex:
nbs = vert2rm.get_all_neighbours()
vtx0 = vertex_map (nbs [0]) # = vertex A
vtx1 = vertex_map (nbs [1]) # = vertex B
vtx0.replace_neighbour (vert2rm, nbs [1])
vtx1.replace_neighbour (vert2rm, nbs [0])

edges2rm = vert2edge(b) = [2, 5, 3, 6]
newe = 8
if (vert2rm.is_double)
    newe = c (newe, newe++) # [8, 9]
for ne in newe:
    # insert in edge first
    for e in edges2rm:
        if e.from == nbs [0]:
            if edge(e).old.size () > 0:
                edge(newe).old = edge(e).old
            else
                edge(newe).old = e
            edges2rm.erase (e)
            vert2edge(vert2rm).erase (e)
            v = vertmap (nbs [0])
            v.in.replace (e, ne)
            vertmap (nbs [0]) = v
            stop
    # ne = 8: e = 2; edge2rm = [5, 3, 6]; edges(8).old = 2
    # ne = 9: e = 6; edge2rm = [5]; edges(9).old = 6

    # then out edge:
    for e in edges2rm:
        if e.to == nbs [0]:
            if edge(e).old.size() > 0:
                edge(newe).old = c (edge(newe).old, edge(e).old)
            else:
```

```
            edge(newe).old = c (edge(newe).old, e)
        edges2rm.erase (e)
        vert2edge(vert2rm).erase (e)
        v = vertmap (nbs [0])
        v.out.replace (e, ne)
        vertmap (nbs [0]) = v
        stop
# ne = 8: e = 3; edge2rm = [5, 6]; edges(8).old = [2, 3]
# ne = 9: e = 5; edge2rm = []; edges(9).old = [6, 5]
# update vert2edge map:
vert2edge(vert2rm).insert(ne)
```

So replacing vertex `B` gives the following maps:

```
vert2edge(A) = [8, 9]
vert2edge(B) = [8, 9]
vert2edge(C) = [8, 9, 7, 4]
vert2edge(D) = [7, 4]
edge(8).old  = [2, 3]
edge(9).old  = [6, 5]
```

Then consider removal of vertex `C` according to the same pseudo-code, with `vert2rm = C`, `vtx0 = A`, `vtx1 = D`, and `newedgenum = [10, 11]`. The code then iterates through the following values:

```
ne = 10
# in-edge:
e = 8
edge(e).old = [2, 3]
edge(10).old = edge(8).old = [2, 3]
# out-edge:
e = 4
edge(10).old = c(edge(10).old, 4) = [2, 3, 4]

ne = 11
# in-edge:
e = 7
edge(11).old = 7
# out-edge:
e = 9
edge(11).old = c(edge(11).old, edge(e).old) = [7, 6, 5]
```

This finally gives the desired contraction with the following mappings:

```
vert2edge(A, B, C, D) = [10, 11] # all the same
edge(10).old = [2, 3, 4]
edge(11).old = [7, 6, 5]
```

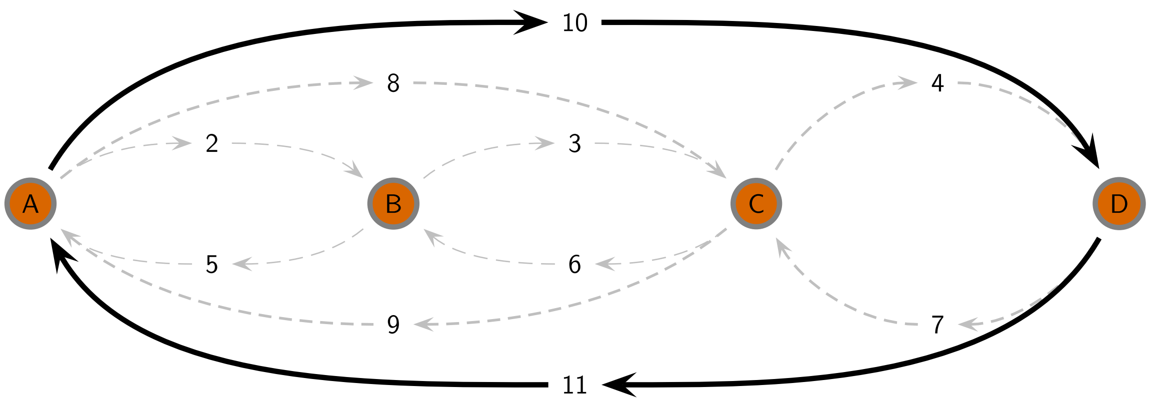

Figure 3: Network of Fig. 1 contracted through removal of vertices B & C.

Graph reconstruction

The maps can then be used to reconstruct a graph. Consider the re-insertion of Vertex B to give the graph of Fig. 4.

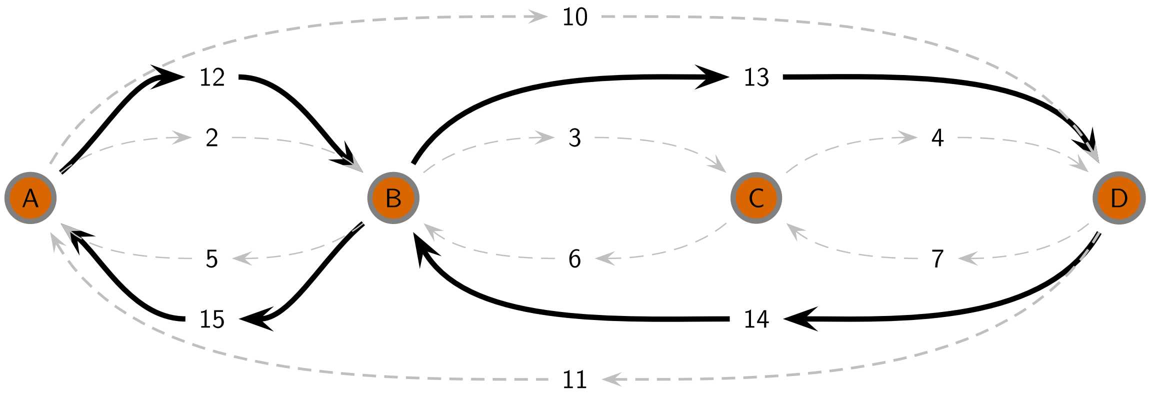

Figure 4: Network of Fig. 1 contracted through removal of vertices B & C.

Vertex B only exists in the vert2edge map as,

```
vert2edge(B) = [10, 11]
```

Then trace both of those edges, extracting the relevant info from the original graph:

```
olde = 10
newe = 12
v2insert = B
e = edge(olde).old = [2, 3, 4]
vlist = e[1].from # = A
vert2edge(vlist[1]).erase(olde)
vert2edge(vlist[1]).insert(newe)
elist = []
d = w = 0
for i in e:
    vlist.push_back(e.to)
    vert2edge(e.to).erase(olde)
    vert2edge(e.to).insert(newe)
    elist.push_back(e)
    d += graph(e).d
    w += graph(e).w
    if (e.to == v2insert)
        vlist1 = vlist
        elist1 = elist
        vlist = e.to
        d1 = d
        w1 = w
        d = w = 0
        edge(newe).old = elist
        elist = []
        newe++
        vert2edge(v2inert).insert(newe)
edge(newe).old = elist
edge(olde).erase
```

A first pass for edge#10 will give the following values

```
newe = 12
vert2edge(A) = [10, 11] -> [12, 11]
vlist = [A]
e = edge(10).old = [2, 3, 4]
(for i in e:)

# e = 2
vlist = [A, B]
vert2edge(B) = [10, 11] -> [12, 11]
```

```
elist = 2
edge(12).old = 2
vlist1 = vlist # = [A, B]
elist1 = elist # = 2
vlist = [B]
elist = []
newe = 13
vert2edge(B) = [12, 11, 13]

# e = 3
vlist = [B, C]
vert2edge(C) = [10, 11] -> [13, 11]
elist = 3

# e = 4
vlist = [B, C, D]
vert2edge(D) = [10, 11] -> [13, 11]
elist = [3, 4]

# out of loop so
edge(13).old = [3, 4]
```

This gives the following:

```
vert2edge(A) = [12, 11]
vert2edge(B) = [12, 11, 13]
vert2edge(C, D) = [13, 11]
edge(12).old = [2]
edge(13).old = [3, 4]
```

Subsequently tracing the second edge#11 with `edge(11).old = [7, 6, 5]` gives,

```
newe = 14
vert2edge(D) = [13, 11] -> [13, 14]
e = [7, 6, 5]
vlist = [D]
(for i in e:)

# e = 7
vlist = [D, C]
vert2edge(C) = [13, 11] -> [13, 14]
elist = 7

# e = 6
vlist = [D, C, B]
elist = [7, 6]
vert2edge(B) = [12, 11, 13] -> [12, 13, 14]
edge(14).old = [7, 6]
vlist1 = vlist # = [D, C, B]
elist1 = elist # = [7, 6]
vlist = [B]
elist = []
newe = 15
vert2edge(B) = [12, 13, 14, 15]

# e = 5
vlist = [B, A]
elist = 5
vert2edge(A) = [12, 11] -> [12, 15]

# out of loop so
edge(15).old = [5]
```

This gives the following final mappings:

```
vert2edge(A)    = [12, 15]
vert2edge(B)    = [12, 13, 14, 15]
vert2edge(C, D) = [13, 14]
edge(12).old = [2]
```

```
edge(13).old = [3, 4]
edge(14).old = [7, 6]
edge(15).old = [5]
```

And that's it!