5.1 Kreis finden (topologisches Sortieren)

Phase 1: lösche Knoten mit $indeg = 0$ bis keine mehr existieren

```
stack<node> ZERO;
node v;

forall_nodes(v,G) {
  if (G.indeg(v) == 0) ZERO.push(v);
}

while (!ZERO.empty())
{ node v = ZERO.pop();
  edge e;
  forall_out_edges(e,v) {
     node w = G.target(e);
     G.del_edge(e);
     if (G.indeg(w) == 0) ZERO.push(w);
  }
  G.del_node(v);
}
```

Phase 2: lösche Knoten mit $outdeg = 0$ bis keine mehr existieren

```
forall_nodes(v,G) {
  if (G.outdeg(v) == 0) ZERO.push(v);
}

while (!ZERO.empty())
{ node v = ZERO.pop();
  edge e;
  forall_in_edges(e,v) {
     node u = G.source(e);
     G.del_edge(e);
     if (G.outdeg(u) == 0) ZERO.push(u);
  }
  G.del_node(v);
}
```

Phase 3: finde einen Kreis (`list<node> cycle`) in G startend in beliebigem Knoten v

```
  if (G.empty()) {
    cout << "G ist azyklisch" << endl;
    return;
  }

// Der Graph ist zyklisch (G ist der Rest, der nur aus Kreisen besteht)
// Starte in beliebigem Knoten v und durchlaufe ausgehende Kanten
// bis wir wieder bei v ankommen (loesche die Kanten, damit jede nur
// einmal benutzt wird).

  node v = G.first_node();
  list<node> cycle;
  cycle.append(v);

  node u = v;
  do { edge e = G.first_out_edge(u);
       u = G.target(e);
       cycle.append(u);
       G.del_edge(e);
  } while (u != v);
```

5.2 DIJKSTRA mit pred-Verweisen

```
void DIJKSTRA(const graph_t& G, node s, const edge_array<int>& cost,
                                        node_array<int>& dist,
                                        node_array<edge>& pred)
{
  priority_queue<node,int> PQ;
  node_array<pq_item> I(G);

  node v;
  forall_nodes(v,G)
  { dist[v] = MAXINT;
    pred[v] = NULL;
    I[v] = NULL;
   }

  dist[s] = 0;
  I[s] = PQ.insert(s,0);

  while (!PQ.empty())
  { node u = PQ.del_min();
    int du = dist[u];
    edge e;
    forall_out_edges(e,u)
    { node v = G.target(e);
      int c = du + cost[e];
      if (c < dist[v])
      { if (dist[v] == MAXINT)
          I[v] = PQ.insert(v,c);
        else
           PQ.decrease_p(I[v],c);
        dist[v] = c;
        pred[v] = e;
       }
    }
  }

}
```

5.3 DFS-Nummerierung

```
dfsnum[v] = 1, ... n  Reihenfolge der rekursiven dfs-Aufrufe

dfsnum[v] == 0: v noch nicht besucht


// rekursive dfs-Funktion

void dfs(const graph& G, node v, node_array<int>& dfsnum, int& count)
{ dfsnum[v] = ++count;
  edge e;
  forall_out_edges(e,v) {
     node w = G.target(e);
     if (dfsnum[w] == 0) dfs(G,w,dfsnum,count);
  }
}


// Hauptprogramm

node_array<int> dfsnum(G,0);
int count = 0;
forall_nodes(v,G) {
  if (dfsnum[v] == 0) dfs(G,v,dfsnum,count);
}
```