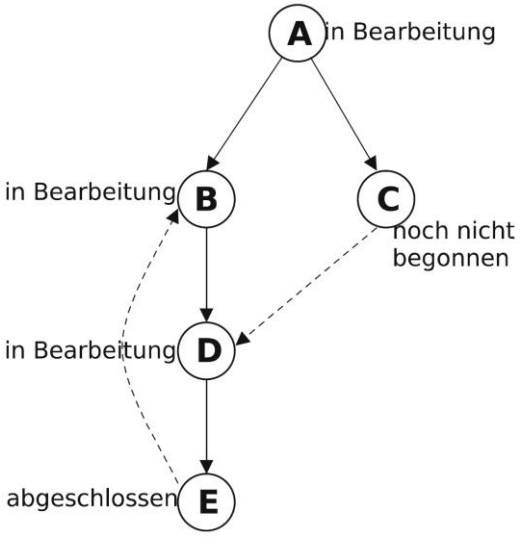
[A10] Zyklen Tiefensuche

Grafische Beschreibung



Textuelle Beschreibung

Bei diesem Beispiel kommt eine, wie bereits im Handout A06 und A09 beschriebene, Tiefensuche zum Einsatz. Jedoch wird bei dieser Übung im Vergleich zu den beiden vorherigen Beispielen mittels eines Zyklus, falls vorhanden, gearbeitet.

Wie in der Grafik gut dargestellt geht es bei einer Zyklischen Tiefensuche nicht primär den besten Weg zu finden, sondern herauszufinden, ob ein Zyklus vorhanden ist und diesen dann, auch wie in der Angabe gefordert, auszugeben. Abhängig, ob ein Graph gerichtet oder ungerichtet ist kommen weitere Faktoren hinzu. So kann in einem gerichteten Graph nur eine Rückwärtskante wie z.B.: von E →B einen Zyklus verursachen.

Laufzeit

Durchschnittsfall: $O(|V| + |E|)$
Wobei $O(|E|)$ zwischen $O(V^2)$ und $O(1)$ wandert

Wichtige Elemente

```
public List<Integer> getCycle() {

   if (graph.numVertices() <= 2 && !graph.isDirected())            // no cycle possible if
vertices count smaller or equal to 2 if graph is undirected
      return null;

   visited = new boolean[graph.numVertices()];                      // set size of visited
array - per default all values are set to false

   for (int i = 0; i < graph.numVertices(); i++) {                   // loop through all
vertices in graph
      if (!visited[i]) {                                        // check if vertex has not been
visited yet
         cycleInGraph = recursiveGetCycle(i, -1);       // call of recursive method
         if (cycleInGraph != null) {                            // cycle has been found
            //System.out.println("method getCycle: " + cycleInGraph);
            return cycleInGraph;
         }
      }
   }
   return null;                                                 // no cycle found -> return null
}
/**
 * Rekurisve Funktion zum Suchen des Zyklus
 * @param vertex Aktueller Knoten
 * @param predecessor Vorg?nger
 * @return Zyklus oder null
 */


private List<Integer> recursiveGetCycle(int vertex, int predecessor) {

   cycleInGraph = new ArrayList<>();
   if (predecessors.containsKey(vertex)) {       // check if vertex is predecessor
      Integer tmp = vertex;              // store vertex as temporary variable
                                    // wrapper class for int needed for "!= null" condition
      while (tmp != null){               // loop through predecessors
         cycleInGraph.add(tmp);          // add node to cycle list
         tmp = predecessors.get(tmp);      // set new predecessor based on current predecessor
      }

      cycleInGraph.add(vertex);           // add vertex as last node to list -> first and last
must be equal

      //System.out.println("cycleInGraph" + cycleInGraph);
      return cycleInGraph;               // return list with nodes
   }

   if (visited[vertex])       // check if was node already visited
      return null;          // if it was return from method

   visited[vertex] = true;                // mark node as visited with "true"

   predecessors.put(predecessor, vertex);     // add predecessor to list

   for (WeightedEdge edge : graph.getEdges(vertex)) {                    // loop through all
edges of vertex from current recursive call
      if (!(edge.to_vertex == predecessor && !graph.isDirected())) {        // check if edge
to vertex equals the predecessor from recursive call
                                                                  // and graph is not directed is not
true
         cycleInGraph = recursiveGetCycle(edge.to_vertex, vertex);     // recursive call based
on vertex which is a predecessor of next vertex (=to_vertex)
         if (cycleInGraph != null) {                              // cycle has been found
            //System.out.println("cycleInGraph" + cycleInGraph);
            return cycleInGraph;
         }
      }

   }

   predecessors.remove(predecessor); // remove predecessor (=backtracking)

   return null; // no cycle found -> return null
}
```