Ausarbeitung 05

Niklas Vest

November 12, 2018

1 Abstrakter Datentyp "Gerichteter Graph"

1.1 Lösungsidee

1.1.1 ADT

In einem Header *directed_graph.h* befinden sich die Schnittstellen für den abstrakten Datentyp "Gerichteter Graph". In einem separaten Header *graph_types.h* sind dann Typen und Typ-Aliases beschrieben. In *CmakeLists.txt* werden dann zwei Build-Targets angelegt. Eines von beiden wird mit einer extra Preprozessor Definition $UE05\_USE\_LIST$ gebaut, anhand deren dann zwischen Matrix- und Listen-Implementierung ausgewählt wird. Im code befinden sich Prüfungen auf die Existenz dieser Definition: Ist die Definition vorhanden, wird der Text aus $directed\_graph_mat.c$ entfernt und der Linked verbindet Aufrufe an die Schnittstellen des Graphentyps mit den Implementierungen in *directed_list.c*. Wenn $UE05\_USE\_LIST$ nicht gesetzt ist, wird die Matrix Implementierung verwendet.

1.1.2 Matrix

Bei der Matrix Implementierung besteht werden für die Informationsverwaltung bzgl. ein- und ausgehender Kanten eine Ajazenzmatrix verwendet. Diese ist als eindimensionales Array von *bool*s realisiert. Die Knoten des Graphen werden in einer einfach verketteten Liste gespeichert.

1.1.3 Liste

Die zweite Implementierung des ADT verwendet Listen, welche Aufschluss über ausgehende Kanten geben. Eingehende Kanten können dadurch nur mühselig berechnet werden, in dem man jede Adiazenzliste jedes Knotens betrachtet und nach einem bestimmten Zielknoten sucht.

Um eine zweite Implementierung einer einfach verketteten Liste zu vermeiden, verwende ich zur Speicherung von Knoten keine weitere Liste sondern eine minimale Vektor Implementierung. Ein Feld an Knoten - deren Index im Feld zugleich ihre ID ist - wird bei Überfüllung mit doppelter Kapazität neu angelegt.

Anmerkung: Man hätte mit *void∗* eine gewisse ”Generizität” erlangen können, man wäre dann aber gezwungen bei jedem Zugriff auf die Daten des knotens einen type cast durchzuführen und das ist nicht nur unsauber sondern auch unsicher.

1.2 Implementierung

Listing 1: *main.c*

```
1  #include "directed_graph.h"
2  #include "unic.h"
3
4  int main()
5  {
6      unic_init();
7
8      // create graph
9      graph_ptr pgraph = create_graph();
10     unic_ass_true(pgraph != NULL, "Graph creation");
11     unic_ass_eq_i((int) nr_of_nodes(pgraph), 0, "No nodes on new graph");
12     unic_ass_eq_i((int) nr_of_edges(pgraph), 0, "No edges on new graph");
13
14     //add nodes
15
16     node_id a = add_graph_node(pgraph, "A");
17     unic_ass_eq_i((int) nr_of_nodes(pgraph), 1, "New node nr after node
        insert");
18     unic_ass_eq_i((int) nr_of_edges(pgraph), 0, "No edges after node insert")
        ;
19
20     node_id b = add_graph_node(pgraph, "B");
21     unic_ass_eq_i((int) nr_of_nodes(pgraph), 2, "New node nr after two node
        inserts");
22     unic_ass_eq_i((int) nr_of_edges(pgraph), 0, "No edges after two node
        inserts");
23
24     node_id c = add_graph_node(pgraph, "C");
25
26     node_id d = add_graph_node(pgraph, "D");
27
28     node_id e = add_graph_node(pgraph, "E");
29     unic_ass_eq_i((int) nr_of_nodes(pgraph), 5, "New node nr after all node
        insert");
30     unic_ass_eq_i((int) nr_of_edges(pgraph), 0, "No edges after all node
        inserts");
31
32     // edges from A
33
34     add_graph_edge(pgraph, a, b);
35     unic_ass_eq_i(5, (int) nr_of_nodes(pgraph), "No new node nr after edge
        insert");
36     unic_ass_eq_i(1, (int) nr_of_edges(pgraph), "New edge nr after edge
        insert");
37
38     add_graph_edge(pgraph, a, c);
```

```
39      unic_ass_eq_i(5, (int) nr_of_nodes(pgraph), "No new node nr after two
         edge inserts");
40      unic_ass_eq_i(2, (int) nr_of_edges(pgraph), "New edge nr after two edge
         inserts");
41
42      add_graph_edge(pgraph, a, d);
43
44      add_graph_edge(pgraph, a, e);
45      unic_ass_eq_i(5, (int) nr_of_nodes(pgraph), "No new node nr after all
         edge inserts");
46      unic_ass_eq_i(4, (int) nr_of_edges(pgraph), "New edge nr after all edge
         inserts");
47
48      unic_ass_eq_i(4, (int) node_out_degree(pgraph, a), "Correct outgoing node
          degree");
49
50      // edges from B
51      add_graph_edge(pgraph, b, d);
52      unic_ass_eq_i(5, (int) nr_of_nodes(pgraph), "No new node nr after new
         edge insert from different node");
53      unic_ass_eq_i(5, (int) nr_of_edges(pgraph), "New edge nr after new edge
         insert from different node");
54
55      unic_ass_eq_i(2, (int) node_in_degree(pgraph, d), "Correct incoming node
         degree");
56
57      // edges from C
58      add_graph_edge(pgraph, c, b);
59      add_graph_edge(pgraph, c, e);
60
61      // print dis boi once
62      print_graph(pgraph);
63
64      // remove an existing edge
65      remove_graph_edge(pgraph, b, d);
66      unic_ass_eq_i(5, (int) nr_of_nodes(pgraph), "Unchanged number of node
         after edge removal");
67      unic_ass_eq_i(6, (int) nr_of_edges(pgraph), "Decreased number of edges
         after edge removal");
68
69      // remove none existend edge
70      remove_graph_edge(pgraph, b, a);
71      unic_ass_eq_i(5, (int) nr_of_nodes(pgraph), "Unchanged number of nodes
         after nonexistent edge removal");
72      unic_ass_eq_i(6, (int) nr_of_edges(pgraph), "Unchanged number of edges
         after nonexistent edge removal");
73
74      // remove existing node
75      remove_graph_node(pgraph, c);
76      unic_ass_eq_i(4, (int) nr_of_nodes(pgraph), "Decreased number of nodes
         after node removal");
77      unic_ass_eq_i(3, (int) nr_of_edges(pgraph), "Decreased number of edges
         after node removal");
78
79      unic_ass_eq_i(3, (int) node_out_degree(pgraph, a), "Correct outgoing node
          degree after manipulation");
80      unic_ass_eq_i(1, (int) node_in_degree(pgraph, e), "Correct incoming node
```

```
        degree after manipulation");
81
82      // remove nonexistent node
83      remove_graph_node(pgraph, 8);
84      unic_ass_eq_i(4, (int) nr_of_nodes(pgraph), "Unchanged number of nodes
         after nonexistent node removal");
85      unic_ass_eq_i(3, (int) nr_of_edges(pgraph), "Unchanged number of edges
         after nonexistent node removal");
86
87      // a second print
88      // note: print_graph prints only nodes
89      // connected by an edge
90      print_graph(pgraph);
91
92      delete_graph(&pgraph);
93      unic_ass_true(pgraph == NULL, "Graph is null pointer after delete");
94      return unic_get_results();
95  }
```

Listing 2: *directed_graph.h*

```
1   #ifndef UE05_DIRECTED_GRAPH_H
2   #define UE05_DIRECTED_GRAPH_H
3
4   #include <stdlib.h>
5   #include "graph_types.h"
6
7   /**
8    * @return A pointer to a newly created graph.
9    */
10  graph_ptr create_graph();
11
12  /**
13   * Deletes the supplied graph.
14   * @param ppgraph The graph to delete.
15   */
16  void delete_graph(graph_ptr *ppgraph);
17
18  /**
19   * @param pgraph The graph to add a node to.
20   * @param payload The payload of the graph node.
21   * @return The id of the newly created node.
22   * @Note If the supplied graph is NULL, the function
23   *       will still return a theoretically valid id
24   *       but any usage results in undefined behaviour.
25   */
26  node_id add_graph_node(graph_ptr pgraph, node_payload payload);
27
28  /**
29   * Removes the node with the specified id and all edges
30   * associated with it from the graph.
31   * If the node does not know about the node with said id,
32   * this function does nothing.
33   * @param pgraph The graph to remove the node from.
34   * @param id The id of the node to delete.
35   */
36  void remove_graph_node(graph_ptr pgraph, node_id id);
37
```

```
38 /**
39  * Adds a directed edge to the graph, starting from the
40  * supplied source node and ending at the destination node.
41  * @param pgraph The graph to add the edge to.
42  * @param from The source node for the edge.
43  * @param to The destination node for the edge.
44  */
45 void add_graph_edge(graph_ptr pgraph, node_id from, node_id to);
46
47 /**
48  * Removes the edge going from <i>from</i> to <i>to</i>.
49  * @param pgraph The graph to remove the edge from.
50  * @param from The source node of the edge.
51  * @param to The destination node of the edge.
52  */
53 void remove_graph_edge(graph_ptr pgraph, node_id from, node_id to);
54
55 /**
56  * @param cpgraph The graph to query.
57  * @return The number of nodes in the graph.
58  */
59 size_t nr_of_nodes(cgraph_ptr cpgraph);
60
61 /**
62  * @param pgraph The graph to query.
63  * @return The number of edges in the graph.
64  */
65 size_t nr_of_edges(cgraph_ptr pgraph);
66
67 /**
68  * Prints all relevant nodes. (i.e. nodes
69  * that have outgoing or incoming directed
70  * edges.)
71  * @param cpgraph The graph to print.
72  */
73 void print_graph(cgraph_ptr cpgraph);
74
75 /**
76  * A simple iterator function which executes the
77  * supplied consumer function for each edge in
78  * the graph (with the respective edge as
79  * argument).
80  * @param cpgraph The graph to iterate.
81  * @param consume_edge_f The consumer function to use.
82  */
83 void for_each_edge(cgraph_ptr cpgraph, graph_edge_consumer consume_edge_f);
84
85 /**
86  * @param cpgraph The graph to query.
87  * @param id The id of the node to fetch information from.
88  * @return The number of outgoing edges.
89  */
90 size_t node_out_degree(cgraph_ptr cpgraph, node_id id);
91
92 /**
93  * @param cpgraph The graph to query.
94  * @param id The id of the node to fetch information from.
```

```
95  * @return The number of incoming edges.
96  */
97 size_t node_in_degree(cgraph_ptr cpgraph, node_id id);
98
99 /**
100  * @param cpgraph The graph from which to fetch 'em nodes.
101  * @return An array containing all nodes of the graphs
102  *         including their respective direct dependencies.
103  */
104 conscious_node *get_conscious_nodes(cgraph_ptr cpgraph);
105
106 /**
107  * @param cpgraph The graph to query.
108  * @param id The id of the node to fetch.
109  * @return The node within the supplied graph with the
110  *          specified id. If the graph does not contain
111  *          such a node, NULL is returned.
112  */
113 graph_node_ptr get_graph_node_by_id(cgraph_ptr cpgraph, node_id id);
114
115 #endif //!UE05_DIRECTED_GRAPH_H
```

Listing 3: *graph_types.h*

```
1  #ifndef UE05_GRAPH_TYPES_H
2  #define UE05_GRAPH_TYPES_H
3
4  #include <stdlib.h>
5
6  /**
7   * The data associated with graph nodes.
8   */
9  typedef const char* node_payload;
10
11 /**
12  * The identifying descriptor of graph nodes.
13  */
14 typedef unsigned node_id;
15
16 /**
17  * A node within a {@link graph}.
18  */
19 typedef struct {
20     /**
21      * The unique node descriptor.
22      */
23     node_id id;
24
25     /**
26      * The data associated with a node.
27      */
28     node_payload payload;
29 } graph_node;
30 typedef graph_node *graph_node_ptr;
31
32 /**
33  * Represents a graph nodes dependencies.
34  * (i. e. all nodes that have an outgoing
```

```
35  * edge landing at said node.)
36  */
37 typedef node_id *dependencies;
38
39 /**
40  * A node which is aware of its dependencies.
41  */
42 typedef struct {
43     /**
44      * All information about the node itself.
45      */
46     graph_node data;
47
48     /**
49      * The dependencies of the node.
50      */
51     dependencies dependency_arr;
52
53     /**
54      * Have a guess what this describes.
55      */
56     size_t nr_of_dependencies;
57 } conscious_node;
58
59 /**
60  * A directed graph.
61  */
62 typedef struct graph graph;
63 typedef struct graph *graph_ptr;
64 typedef const struct graph *cgraph_ptr;
65
66 /**
67  * Functions of this type can be used with the graphs
68  * for_each_edge function to achieve some sort of
69  * iterative behaviour without the need of a separate
70  * iterator class for each graph implementation.
71  *  Objects of this function type are supposed
72  *  to be supplied with the source and destination node
73  *  of a directed edge.
74  */
75 typedef void(*graph_edge_consumer)(graph_node_ptr from, graph_node_ptr to);
76
77 #endif //!UE05_GRAPH_TYPES_H
```

Listing 4: *directed_graph_list.c*

```
1  // warning: ISO C forbids an empty translation unit
2  typedef int i_am_a_number_making_the_compiler_happy_because_the_pedantic\
3  _compile_option_warns_about_empty_translation_units;
4
5  #ifdef UE05_USE_LIST
6
7  #include <stdbool.h>
8  #include <memory.h>
9  #include "directed_graph.h"
10 #include "node_list.h"
11
12 #define INITIAL_VECTOR_SIZE 5
```

```
13
14 /**
15  * A dynamically growing array of list node pointers.
16  */
17 typedef list_node_ptr *list_node_ptr_vec;
18
19 struct graph {
20     /**
21      * A vector containing all nodes of the graph.
22      */
23     list_node_ptr_vec nodes;
24
25     /**
26      * The current size of the vector.
27      */
28     size_t vec_size;
29 };
30
31 /**
32  * @param payload The payload of the node.
33  * @param id The id of the node.
34  * @return A pointer to the newly created graph node.
35  */
36 static graph_node_ptr create_graph_node(node_payload payload, node_id id);
37
38 /**
39  * @param pgraph The graph for which to fetch the first
40  *               free node it.
41  * @return The first available node id for the
42  *         supplied graph. If the graphs adjacency matrix
43  *         is full, it is resized.
44  */
45 static node_id generate_id(graph_ptr pgraph);
46
47 /**
48  * Resizes the supplied graphs node vector
49  * by doubling its capacity.
50  * @param pgraph The graph of which the node vector
51  *                 shall be resized.
52  */
53 static void grow_node_vector(graph_ptr pgraph);
54
55 /**
56  * @param pgraph The graph to query.
57  * @param id The id of the node to search for.
58  * @return True if the node exists, false
59  *         otherwise.
60  */
61 static bool node_exists(cgraph_ptr pgraph, node_id id);
62
63 /**
64  * Deletes the node with the specified id.
65  * @param pgraph The graph from which to delete the node.
66  * @param id The id of the node to delete.
67  */
68 static void delete_graph_node(graph_ptr pgraph, node_id id);
69
```

```
70 graph_ptr create_graph()
71 {
72     graph_ptr pgraph = (graph_ptr) malloc(sizeof(graph));
73     pgraph->nodes = (list_node_ptr_vec) malloc(sizeof(list_node_ptr) *
        INITIAL_VECTOR_SIZE);
74     for (size_t i = 0; i < INITIAL_VECTOR_SIZE; ++i) {
75         pgraph->nodes[i] = NULL;
76     }
77     pgraph->vec_size = INITIAL_VECTOR_SIZE;
78     return pgraph;
79 }
80
81 void delete_graph(graph_ptr *ppgraph)
82 {
83     if (ppgraph != NULL && *ppgraph != NULL) {
84         graph_ptr pgraph = *ppgraph;
85         // if pgraph is not null, pgraph->nodes
86         // is guaranteed to not be null because it's
87         // a vector which is first allocated during
88         // graph creation!
89         for (node_id id = 0; id < pgraph->vec_size; ++id) {
90             if (node_exists(pgraph, id)) {
91                 delete_graph_node(pgraph, id);
92             }
93         }
94         free(pgraph->nodes);
95         free(pgraph);
96         *ppgraph = NULL;
97     }
98 }
99
100 void delete_graph_node(graph_ptr pgraph, node_id id)
101 {
102     if (node_exists(pgraph, id)) {
103         free((char *) pgraph->nodes[id]->data->payload);
104         free(pgraph->nodes[id]->data);
105         delete_list(&(pgraph->nodes[id]));
106     }
107 }
108
109 node_id add_graph_node(graph_ptr pgraph, node_payload payload)
110 {
111     node_id id = 0;
112     if (pgraph != NULL) {
113         id = generate_id(pgraph);
114         graph_node_ptr pnode = create_graph_node(payload, id);
115         // if the next free id is not within the vectors bounds
116         if (pgraph->vec_size <= id) {
117             grow_node_vector(pgraph);
118         }
119         // add the graph node to the vector
120             // the vector contains lists with their head
121             // being the source graph node of an edge
122             // and all ancestors representing the destination
123             // nodes for edges going out from said source node.
124         append_to_list_sorted(&(pgraph->nodes[id]), pnode);
125     }
```

```
126     return id;
127 }
128
129 void remove_graph_node(graph_ptr pgraph, node_id id)
130 {
131     if (node_exists(pgraph, id)) {
132         // iterate all nodes and delete edges to the node with the specified id
133         for (size_t current_id = 0; current_id < pgraph->vec_size; ++
        current_id) {
134             if (node_exists(pgraph, (node_id) current_id)) {
135                 // since the node_id is the index into the vector
136                 remove_graph_edge(pgraph, (node_id) current_id, id);
137             }
138         }
139         delete_graph_node(pgraph, id);
140     }
141 }
142
143 void add_graph_edge(graph_ptr pgraph, node_id from, node_id to)
144 {
145     // if both from and to exist
146     if (node_exists(pgraph, from) && node_exists(pgraph, to)) {
147         // add an entry with TO to the edge-list of FROM
148         graph_node_ptr to_ptr = pgraph->nodes[to]->data;
149         append_to_list_sorted(&(pgraph->nodes[from]), to_ptr);
150     }
151 }
152
153 void remove_graph_edge(graph_ptr pgraph, node_id from, node_id to)
154 {
155     if (node_exists(pgraph, from) && node_exists(pgraph, to)) {
156         // traverse list of edges until the node with the desired ID is reached
157         list_node_ptr prev_edge = pgraph->nodes[from];
158         list_node_ptr curr_edge =prev_edge->next;
159         while (curr_edge != NULL && curr_edge->data->id != to) {
160             prev_edge = curr_edge;
161             curr_edge = curr_edge->next;
162         }
163         if (curr_edge != NULL) {
164             // and delete dat boi
165             delete_list_node_after(prev_edge);
166         }
167     }
168 }
169
170 size_t nr_of_nodes(const cgraph_ptr cpgraph)
171 {
172     size_t nr = 0;
173     if (cpgraph != NULL) {
174         for (size_t i = 0; i < cpgraph->vec_size; ++i) {
175             if (node_exists(cpgraph, (node_id) i)) {
176                 nr += 1;
177             }
178         }
179     }
180     return nr;
181 }
```

```
182
183 size_t nr_of_edges(const cgraph_ptr pgraph)
184 {
185     size_t nr = 0;
186     if (pgraph != NULL) {
187         for (size_t i = 0; i < pgraph->vec_size; ++i) {
188             if (node_exists(pgraph, (node_id) i)) {
189                 // for each node in the vector
190                 list_node_ptr curr_edge = pgraph->nodes[i]->next;
191                 while (curr_edge != NULL) {
192                     // add +1 to the total sum for each
193                     // list node after the head
194                     nr += 1;
195                     curr_edge = curr_edge->next;
196                 }
197             }
198         }
199     }
200     return nr;
201 }
202
203 void for_each_edge(const cgraph_ptr cpgraph, graph_edge_consumer
      consume_edge_f)
204 {
205     if (cpgraph != NULL) {
206         // for each source node
207         for (size_t i = 0; i < cpgraph->vec_size; ++i) {
208             list_node_ptr curr_src_node = cpgraph->nodes[i];
209             if (curr_src_node != NULL) {
210                 graph_node_ptr src_graph_node = curr_src_node->data;
211                 list_node_ptr curr_dest_node = curr_src_node->next;
212                 // combine it with each of its destination nodes
213                 while (curr_dest_node != NULL) {
214                     // and invoke the callback
215                     consume_edge_f(src_graph_node, curr_dest_node->data);
216                     curr_dest_node = curr_dest_node->next;
217                 }
218             }
219         }
220     }
221 }
222
223 size_t node_out_degree(const cgraph_ptr cpgraph, node_id id)
224 {
225     size_t degree = 0;
226     if (node_exists(cpgraph, id)) {
227         // first node is source node, ancestors are edge destinations
228         list_node_ptr edge_list = cpgraph->nodes[id]->next;
229         degree = get_list_size(edge_list);
230     }
231     return degree;
232 }
233
234 size_t node_in_degree(const cgraph_ptr cpgraph, node_id id)
235 {
236     size_t degree = 0;
237     // if the destination is valid
```

```
238     if (node_exists(cpgraph, id)) {
239         // for each node in the vector
240         for (size_t i = 0; i < cpgraph->vec_size; ++i) {
241             if (node_exists(cpgraph, (node_id) i)) {
242                 // check if its edge list contains
243                 // the node with the specified id
244                 list_node_ptr curr_dest_node = cpgraph->nodes[i]->next;
245                 bool has_edge_to_id = false;
246                 while (curr_dest_node != NULL && !has_edge_to_id) {
247                     has_edge_to_id = curr_dest_node->data->id == id;
248                     curr_dest_node = curr_dest_node->next;
249                 }
250                 if (has_edge_to_id) {
251                     degree += 1;
252                 }
253             }
254         }
255     }
256     return degree;
257 }
258
259 // ------------------------------------ TRANSLATION
       -UNIT-LOCAL FUNCTIONS
       ------------------------------------ //
260
261 graph_node_ptr create_graph_node(node_payload payload, node_id id)
262 {
263     graph_node_ptr pnode = (graph_node_ptr) malloc(sizeof(graph_node));
264     char *str_copy = (char *) malloc(sizeof(char) * strlen(payload) + 1);
265     strcpy(str_copy, payload);
266     pnode->payload = str_copy;
267     pnode->id = id;
268     return pnode;
269 }
270
271 node_id generate_id(graph_ptr pgraph)
272 {
273     // if the graph is null, return 0
274     node_id id = 0;
275     if (pgraph != NULL) {
276         // search the vector for the first entry
277         // that is not NULL. if the vector is full,
278         // return an id outside the bounds of the vector
279         while (id < pgraph->vec_size && pgraph->nodes[id] != NULL) {
280             id += 1;
281         }
282     }
283     return id;
284 }
285
286 void grow_node_vector(graph_ptr pgraph)
287 {
288     if (pgraph != NULL) {
289         size_t new_length = pgraph->vec_size * 2;
290         pgraph->nodes = (list_node_ptr_vec) realloc(pgraph->nodes, sizeof(
       list_node_ptr) * new_length);
291         for (size_t i = 0; i < pgraph->vec_size; ++i) {
```

```
292             // also initialize new items to NULL
293             pgraph->nodes[pgraph->vec_size + i] = NULL;
294         }
295         pgraph->vec_size = new_length;
296     }
297 }
298
299 bool node_exists(const cgraph_ptr pgraph, node_id id)
300 {
301     return pgraph != NULL && id < pgraph->vec_size && pgraph->nodes[id] !=
      NULL;
302 }
303
304 #endif
```

Listing 5: *directed_graph$_m$at.c*

```
1  // warning: ISO C forbids an empty translation unit
2  typedef int i_am_a_number_making_the_compiler_happy_because_the_pedantic\
3  _compile_option_warns_about_empty_translation_units;
4
5  #ifndef UE05_USE_LIST
6
7  #include <stdlib.h>
8  #include <stdbool.h>
9  #include <memory.h>
10 #include "../adt/graph_types.h"
11 #include "../adt/directed_graph.h"
12 #include "../adt/node_list.h"
13
14 /**
15  * The type of the elements in the
16  * adjacency matrix.
17  */
18 typedef bool adjacenty_mat_type;
19
20 /**
21  * A matrix representing adjacency relations.
22  */
23 typedef adjacenty_mat_type *adjacency_mat;
24
25 struct graph {
26     /**
27      * A matrix representing relationships
28      * between all nodes of the graph.
29      */
30     adjacency_mat edges;
31
32     /**
33      * A single linked list of graph nodes.
34      */
35     list_node_ptr nodes;
36
37     /**
38      * The current dimension of the edge matrix.
39      */
40     size_t mat_side_len;
41 };
```

```
42
43 /**
44  * @param pgraph The graph for which to fetch the first
45  *               free node it.
46  * @return The first available node id for the
47  *         supplied graph. If the graphs adjacency matrix
48  *         is full, it is resized.
49  */
50 static node_id generate_id(graph_ptr pgraph);
51
52 /**
53  * A function <i>f : R<sup>2</sup> --> R</i>
54  * which maps the 2D indices <i>from</i> and
55  * <i>to</i> to a 1D array index.
56  * @param side_length The side length of the virtual matrix.
57  * @param from The row to target.
58  * @param to The column to target.
59  * @return The projected 1D array index.
60  */
61 static size_t calc_mat_index(size_t side_length, node_id from, node_id to);
62
63 /**
64  * Sets the specified edge value for the edge from node
65  * <i>from</i> to node <i>to</i>.
66  * @param pgraph The graph to set the edge value for.
67  * @param from The source node of the edge.
68  * @param to The destination node of the edge.
69  * @param edge_flag The value to set. <i>true</i> adds the
70  *                  edge, <i>false</i> removes it.
71  */
72 static void set_edge_value(graph_ptr pgraph, node_id from, node_id to, bool
      edge_flag);
73
74 /**
75  * @param payload The payload of the node.
76  * @param id The id of the node.
77  * @return A pointer to the newly created graph node.
78  */
79 static graph_node_ptr create_graph_node(node_payload payload, node_id id);
80
81 /**
82  * Increases the supplied graphs edge matrix by one
83  * row and one column.
84  * @param pgraph The graph to manipulate.
85  */
86 static void enlarge_matrix(graph_ptr pgraph);
87
88 /**
89  * @param cpgraph The graph to query.
90  * @param id The id to search for.
91  * @return True if there is a node with the
92  *          specified ID in the supplied graph.
93  */
94 static bool node_exists(cgraph_ptr cpgraph, node_id id);
95
96 /**
97  * @param cpgraph The graph to query
```

```
98  * @param source The source node.
99  * @param destination The destination node.
100 * @return True if there is an edge going from the
101 *          supplied source node to the destination node,
102 *          False otherwise.
103 */
104 bool node_has_edge_to(cgraph_ptr cpgraph, node_id source, node_id destination
        );
105
106 graph_ptr create_graph()
107 {
108     graph_ptr pgraph = (graph_ptr) malloc(sizeof(graph));
109     pgraph->mat_side_len = 0;
110     pgraph->edges = NULL;
111     pgraph->nodes = NULL;
112     return pgraph;
113 }
114
115 void delete_graph(graph_ptr *ppgraph)
116 {
117     if (ppgraph != NULL && *ppgraph != NULL) {
118         graph_ptr pgraph = *ppgraph;
119
120         // delete_list only deletes list nodes, not
121         // associated data! We have to do that manually
122         list_node_ptr curr = pgraph->nodes;
123         while (curr != NULL) {
124             free((char *) curr->data->payload);
125             free(curr->data);
126             curr = curr->next;
127         }
128
129         // free remaining memory associated with the graph
130         free(pgraph->edges);
131         delete_list(&(pgraph->nodes));
132         free(pgraph);
133
134         *ppgraph = NULL;
135     }
136 }
137
138 node_id add_graph_node(graph_ptr pgraph, node_payload payload)
139 {
140     node_id id = 0;
141     if (pgraph != NULL) {
142         id = generate_id(pgraph);
143         graph_node_ptr pnode = create_graph_node(payload, id);
144         append_to_list_sorted(&(pgraph->nodes), pnode);
145     }
146     return id;
147 }
148
149 void remove_graph_node(graph_ptr pgraph, node_id id)
150 {
151     if (pgraph != NULL) {
152         list_node_ptr node_to_delete = get_node_by_id(pgraph->nodes, id);
153         if (node_to_delete != NULL) { // if it exists
```

```
154             // remove all edges to the node
155             for (unsigned i = 0; i < nr_of_nodes(pgraph); ++i) {
156                 remove_graph_edge(pgraph, id, i);
157                 remove_graph_edge(pgraph, i, id);
158             }
159
160             list_node_ptr curr = pgraph->nodes;
161
162             // if the node to delete is not the first one
163             if (node_to_delete != curr) {
164                 // go on an epic adventure to find the
165                 // predecessor to dat boi
166                 while (curr->next != node_to_delete) {
167                     curr = curr->next;
168                 }
169
170                 // unchain node_to_delete
171                 curr->next = node_to_delete->next;
172             } else {
173                 // uncahin node_to_delete
174                 pgraph->nodes = node_to_delete->next;
175             }
176
177             // "encapsulate" the node so it
178             // represents an individual list
179             node_to_delete->next = NULL;
180             // and delete it
181             free((char *) node_to_delete->data->payload);
182             free(node_to_delete->data);
183             delete_list(&node_to_delete);
184         }
185     }
186 }
187
188 void add_graph_edge(graph_ptr pgraph, node_id from, node_id to)
189 {
190     set_edge_value(pgraph, from, to, true);
191 }
192
193 void remove_graph_edge(graph_ptr pgraph, node_id from, node_id to)
194 {
195     set_edge_value(pgraph, from, to, false);
196 }
197
198 size_t nr_of_nodes(const cgraph_ptr pgraph)
199 {
200     size_t nr = 0;
201     if (pgraph != NULL) {
202         nr = get_list_size(pgraph->nodes);
203     }
204     return nr;
205 }
206
207 size_t nr_of_edges(const cgraph_ptr pgraph)
208 {
209     size_t nr = 0;
210     if (pgraph != NULL) {
```

```
211         size_t mat_arr_length = pgraph->mat_side_len * pgraph->mat_side_len;
212         for (size_t i = 0; i < mat_arr_length; ++i) {
213             if (pgraph->edges[i]) {
214                 nr += 1;
215             }
216         }
217     }
218     return nr;
219 }
220
221 void for_each_edge(const cgraph_ptr cpgraph, graph_edge_consumer
        consume_edge_f)
222 {
223     if (cpgraph != NULL && consume_edge_f != NULL) {
224         // for each row (source node)
225         for (node_id from = 0; from < cpgraph->mat_side_len; ++from) {
226             list_node_ptr from_ptr = get_node_by_id(cpgraph->nodes, from);
227             // for each column (destination node) of the current row
228             for (node_id to = 0; to < cpgraph->mat_side_len; ++to) {
229                 // if they are connected via an edge
230                 if (cpgraph->edges[calc_mat_index(cpgraph->mat_side_len, from
        , to)]) {
231                     list_node_ptr to_ptr = get_node_by_id(cpgraph->nodes, to)
        ;
232                     // consume it
233                     consume_edge_f(from_ptr->data, to_ptr->data);
234                 }
235             }
236         }
237     }
238 }
239
240 size_t node_out_degree(const cgraph_ptr cpgraph, node_id source)
241 {
242     size_t degree = 0;
243     if (node_exists(cpgraph, source)) {
244         for (node_id i = 0; i < cpgraph->mat_side_len; ++i) {
245             // +1 for each edge in the <source>th row
246             // because the row _i_ represents outgoing
247             // edges to the node with id _i_
248             if (node_has_edge_to(cpgraph, source, i)) {
249                 degree += 1;
250             }
251         }
252     }
253     return degree;
254 }
255
256 size_t node_in_degree(const cgraph_ptr cpgraph, node_id destination)
257 {
258     size_t degree = 0;
259     // if destination is not out of bounds for the matrix
260     if (node_exists(cpgraph, destination)) {
261         // +1 for each edge in the <destination>th row
262         // because the row _i_ represents outgoing
263         // edges to the node with id _i_
264         for (node_id i = 0; i < cpgraph->mat_side_len; ++i) {
```

```
265             if (node_has_edge_to(cpgraph, i, destination)) {
266                 degree += 1;
267             }
268         }
269     }
270     return degree;
271 }
272
273 conscious_node *get_conscious_nodes(const cgraph_ptr cpgraph)
274 {
275     conscious_node *nodes_arr = NULL;
276     if (cpgraph != NULL) {
277         size_t node_count = nr_of_nodes(cpgraph);
278         nodes_arr = (conscious_node *) malloc(sizeof(conscious_node) *
        node_count);
279         for (node_id i = 0; i < node_count; ++i) {
280             if (node_exists(cpgraph, i)) {
281                 conscious_node cn;
282                 cn.data = *(get_graph_node_by_id(cpgraph, i));
283                 cn.dependency_arr = (dependencies) malloc(sizeof(node_id) *
        node_in_degree(cpgraph, i));
284                 cn.nr_of_dependencies = 0;
285                 for (node_id dep = 0; dep < cpgraph->mat_side_len; ++dep) {
286                     if (node_has_edge_to(cpgraph, dep, i)) {
287                         cn.dependency_arr[cn.nr_of_dependencies++] = dep;
288                     }
289                 }
290                 nodes_arr[i] = cn;
291             }
292         }
293     }
294     return nodes_arr;
295 }
296
297 graph_node_ptr get_graph_node_by_id(cgraph_ptr cpgraph, node_id id)
298 {
299     graph_node_ptr pgraph_node = NULL;
300     if (cpgraph != NULL) {
301         list_node_ptr curr = cpgraph->nodes;
302         while (curr != NULL && curr->data->id != id) {
303             curr = curr->next;
304         }
305         if (curr != NULL) {
306             pgraph_node = curr->data;
307         }
308     }
309     return pgraph_node;
310 }
311
312 // ------------------------------------ TRANSLATION
      -UNIT-LOCAL FUNCTIONS
      ------------------------------------ //
313
314 graph_node_ptr create_graph_node(node_payload payload, node_id id)
315 {
316     graph_node_ptr pnode = (graph_node_ptr) malloc(sizeof(graph_node));
317     char *str_cpy = (char *) malloc(sizeof(char) * strlen(payload) + 1);
```

```
318     strcpy(str_cpy, payload);
319     pnode->payload = str_cpy;
320     pnode->id = id;
321     return pnode;
322 }
323
324 node_id generate_id(graph_ptr pgraph)
325 {
326     node_id id = 0;
327     // if the graph is empty or there are no nodes
328     // return 0!
329     if (pgraph != NULL) {
330
331         if (pgraph->edges == NULL) {
332             enlarge_matrix(pgraph);
333             // we do not need to do anything else here;
334             // 0 is the next available index!
335         } else {
336             // the node list is sorted so if the first
337             // node id is not 0 we can immediately return 0.
338             // otherwise we have to search for any "ID gaps"
339             if (pgraph->nodes->data->id == 0) {
340
341                 list_node_ptr curr = pgraph->nodes->next;
342                 while (curr != NULL && curr->data->id == id + 1) {
343                     id += 1; // same as curr = curr->data->id;
344                     curr = curr->next;
345                 }
346
347                 // in both cases (curr == NULL and curr != NULL)
348                 // we have to increment since _id_ still holds
349                 // the last unavailable id before the one that's
350                 // available
351                 id +=1 ;
352
353                 if (curr == NULL) {
354                     // id would now index our matrix out
355                     // of bounds so lets make it BIGGUR
356                     enlarge_matrix(pgraph);
357                 }
358             }
359         }
360     }
361     return id;
362 }
363
364 size_t calc_mat_index(size_t side_length, node_id from, node_id to)
365 {
366     return from * side_length + to;
367 }
368
369 void set_edge_value(graph_ptr pgraph, node_id from, node_id to, bool
        edge_flag)
370 {
371     if (pgraph != NULL) {
372         if (node_exists(pgraph, from) && node_exists(pgraph, to)) {
373             pgraph->edges[calc_mat_index(pgraph->mat_side_len, from, to)] =
```

```
        edge_flag;
374         }
375     }
376 }
377
378 void enlarge_matrix(graph_ptr pgraph)
379 {
380     // prevent enlargement if the matrix is not completely
381     // STUFFED with IDs
382     if (pgraph != NULL && pgraph->mat_side_len == nr_of_nodes(pgraph)) {
383         // allocate new matrix
384         size_t new_mat_side_len = pgraph->mat_side_len + 1;
385         size_t arr_length = new_mat_side_len * new_mat_side_len;
386         adjacency_mat new_mat = (adjacency_mat) malloc(sizeof(
        adjacenty_mat_type) * arr_length);
387
388         if (pgraph->mat_side_len == 0) {
389             new_mat[0] = false;
390         } else {
391             // copy values
392             for (node_id from = 0; from < pgraph->mat_side_len; ++from) {
393
394                 // write old values to the new matrix
395                 for (node_id to = 0; to < pgraph->mat_side_len; ++to) {
396                     size_t old_index = calc_mat_index(pgraph->mat_side_len,
        from, to);
397                     size_t new_index = calc_mat_index(new_mat_side_len, from,
         to);
398                     new_mat[new_index] = pgraph->edges[old_index];
399                 }
400
401                 // set the edge flags for new nodes to 0
402
403                 // graph->mat_side_len is now the last column / row since the
404                 // new side length is now graph->mat_side_len + 1
405                 size_t last_row_index = calc_mat_index(new_mat_side_len, (
        node_id) pgraph->mat_side_len, from);
406                 size_t last_col_index = calc_mat_index(new_mat_side_len, from
        , (node_id) pgraph->mat_side_len);
407                 size_t corner_index = calc_mat_index(new_mat_side_len, (
        node_id) pgraph->mat_side_len,
408                                                      (node_id) pgraph->
        mat_side_len);
409                 new_mat[last_row_index] = false;
410                 new_mat[last_col_index] = false;
411                 new_mat[corner_index] = false;
412             }
413         }
414
415         // delete old matrix and deploy new one
416         free(pgraph->edges);
417         pgraph->edges = new_mat;
418         pgraph->mat_side_len += 1;
419     }
420 }
421
422 bool node_exists(const cgraph_ptr pgraph, node_id id)
```

```
423 {
424     list_node_ptr pgraph_node = NULL;
425     if (pgraph != NULL) {
426         pgraph_node = get_node_by_id(pgraph->nodes, id);
427     }
428     return pgraph_node != NULL;
429 }
430
431 bool node_has_edge_to(const cgraph_ptr cpgraph, node_id source, node_id
        destination)
432 {
433     size_t index = calc_mat_index(cpgraph->mat_side_len, source, destination)
        ;
434     return cpgraph->edges[index];
435 }
436
437 #endif
```

Listing 6: *directed_graph.c*

```
1  // common implementations for both
2  // the adjacency matrix and list
3  // version of the directed graph
4
5  #include "directed_graph.h"
6  #include <stdlib.h>
7  #include <stdio.h>
8  #include <stdbool.h>
9
10 /**
11  * Prints a pair of nodes connected by an edge.
12  * @param from The source node.
13  * @param to The destination node.
14  */
15 static void print_pair(graph_node_ptr from, graph_node_ptr to);
16
17 void print_graph(const cgraph_ptr graph)
18 {
19     if (graph != NULL) {
20         for_each_edge(graph, print_pair);
21     }
22 }
23
24 // ---------------------------------------- TRANSLATION
       -UNIT-LOCAL FUNCTIONS
       ---------------------------------------- //
25
26 void print_pair(graph_node_ptr from, graph_node_ptr to)
27 {
28     if (from != NULL && to != NULL) {
29         printf("( %s ) --> ( %s )\n", from->payload, to->payload);
30     }
31 }
```

Listing 7: *node_list.h*

```
1 #ifndef UE05_NODE_LIST_H
2 #define UE05_NODE_LIST_H
```

```
3
4  #include "graph_types.h"
5  #include <stdlib.h>
6
7  /**
8   * A node of a single linked list.
9   */
10 typedef struct list_node {
11     /**
12      * The information stored in the list node.
13      */
14     graph_node_ptr data;
15
16     /**
17      * The next node in the list.
18      */
19     struct list_node *next;
20 } list_node;
21 typedef list_node *list_node_ptr;
22
23 /**
24  * Recursively deletes a list starting from a given node.
25  * Note: this does not delete the data within the list!
26  * @param plist The list to delete.
27  */
28 void delete_list(list_node_ptr *plist);
29
30 /**
31  * Appends the supplied node to the back of the list.
32  * @param plist The list to append to.
33  * @param data The graph data to add to the appended list node.
34  */
35 void append_to_list_sorted(list_node_ptr *plist, graph_node_ptr data);
36
37 /**
38  * @param list The list to query.
39  * @param id The id of the node to fetch.
40  * @return The list node with the specified id.
41  */
42 list_node_ptr get_node_by_id(list_node_ptr list, node_id id);
43
44 /**
45  * Removes the direct ancestor of <i>plist_node</i>.
46  * @param plist_node The predecessor of the node to delete.
47  */
48 void delete_list_node_after(list_node_ptr plist_node);
49
50 /**
51  * @param list The list to use for evaluation.
52  * @return The number of elements in the list.
53  */
54 size_t get_list_size(list_node_ptr list);
55
56 #endif //!UE05_NODE_LIST_H
```

Listing 8: *node_list.c*

```
1 #include "node_list.h"
```

```
2  #include <stdlib.h>
3  #include <assert.h>
4
5  /**
6   * @param data The data to stuff into the list node.
7   * @return A pointer to the newly created list node.
8   */
9  static list_node_ptr create_list_node(graph_node_ptr data);
10
11 void delete_list(list_node_ptr *plist)
12 {
13     if (plist != NULL) {
14         list_node_ptr node = *plist;
15         if (node != NULL) {
16             list_node_ptr next = node->next;
17             free(node);
18             // point plist to the next node in the list;
19             *plist = next;
20             // delete recursively
21             delete_list(plist);
22         }
23         *plist = NULL;
24     }
25 }
26
27 void append_to_list_sorted(list_node_ptr *plist, graph_node_ptr data)
28 {
29     if (plist != NULL && data != NULL) {
30         list_node_ptr node = create_list_node(data);
31         if (*plist == NULL) {
32             *plist = node;
33         } else {
34             list_node_ptr current = *plist;
35             while (current->next != NULL && current->next->data->id < data->
       id) {
36                 // this should nevur evur heppen
37                 assert(data->id != current->next->data->id);
38                 current = current->next;
39             }
40
41             // if there is an "ID gap", insert inbetween
42             if (current->next != NULL && current->next->data->id > data->id)
       {
43                 node->next = current->next;
44             }
45             current->next = node;
46         }
47     }
48 }
49
50 list_node_ptr get_node_by_id(list_node_ptr list, node_id id)
51 {
52     if (list == NULL || list->data->id > id) {
53         return NULL;
54     } else if (list->data->id == id) {
55         return list;
56     } else {
```

```
57          return get_node_by_id(list->next, id);
58      }
59  }
60
61  void delete_list_node_after(list_node_ptr plist_node)
62  {
63      if (plist_node != NULL && plist_node->next != NULL) {
64          list_node_ptr buffer = plist_node->next->next;
65          free(plist_node->next);
66          plist_node->next = buffer;
67      }
68  }
69
70  size_t get_list_size(list_node_ptr list)
71  {
72      size_t size = 0;
73      if (list != NULL) {
74          size = 1 + get_list_size(list->next);
75      }
76      return size;
77  }
78
79  // ---------------------------------------- TRANSLATION
        -UNIT-LOCAL FUNCTIONS
        ---------------------------------------- //
80
81  list_node_ptr create_list_node(graph_node_ptr data)
82  {
83      list_node_ptr node = (list_node_ptr) malloc(sizeof(list_node));
84      node->next = NULL;
85      node->data = data;
86      return node;
87  }
```

1.3 Tests

Das Tool für Speicheranalyse ”Valgrind” fand keine invaliden Speicheroperationen oder -verluste.

```
/home/niklas/Documents/Github/fh-hgb-ws1819/swo/ue05/cmake-build-debug/adtm
@ Test "Graph creation" succeeded.
@ Test "No nodes on new graph" succeeded.
@ Test "No edges on new graph" succeeded.
@ Test "New node nr after node insert" succeeded.
@ Test "No edges after node insert" succeeded.
@ Test "New node nr after two node inserts" succeeded.
@ Test "No edges after two node inserts" succeeded.
@ Test "New node nr after all node insert" succeeded.
@ Test "No edges after all node inserts" succeeded.
@ Test "No new node nr after edge insert" succeeded.
@ Test "New edge nr after edge insert" succeeded.
@ Test "No new node nr after two edge inserts" succeeded.
@ Test "New edge nr after two edge inserts" succeeded.
@ Test "No new node nr after all edge inserts" succeeded.
@ Test "New edge nr after all edge inserts" succeeded.
@ Test "Correct outgoing node degree" succeeded.
@ Test "No new node nr after new edge insert from different node" succeeded.
@ Test "New edge nr after new edge insert from different node" succeeded.
@ Test "Correct incoming node degree" succeeded.
( A ) ⟶ ( B )
( A ) ⟶ ( C )
( A ) ⟶ ( D )
( A ) ⟶ ( E )
( B ) ⟶ ( D )
( C ) ⟶ ( B )
( C ) ⟶ ( E )
@ Test "Unchanged number of node after edge removal" succeeded.
@ Test "Decreased number of edges after edge removal" succeeded.
@ Test "Unchanged number of nodes after nonexistent edge removal" succeeded.
@ Test "Unchanged number of edges after nonexistent edge removal" succeeded.
@ Test "Decreased number of nodes after node removal" succeeded.
@ Test "Decreased number of edges after node removal" succeeded.
@ Test "Correct outgoing node degree after manipulation" succeeded.
@ Test "Correct incoming node degree after manipulation" succeeded.
@ Test "Unchanged number of nodes after nonexistent node removal" succeeded.
@ Test "Unchanged number of edges after nonexistent node removal" succeeded.
( A ) ⟶ ( B )
( A ) ⟶ ( D )
( A ) ⟶ ( E )
@ Test "Graph is null pointer after delete" succeeded.
================
UNIC ran 30 tests.
30 tests succeeded.
================

Process finished with exit code 0
```

Figure 1: Resultat von *adt/main.c*

2 Topologisches Sortieren

2.1 Lösungsidee

Zum topologischen Sortieren habe ich den Algorithmus von Kahn herangezogen: Alle Knoten eines graphen, die keine Abhängigkeiten haben (i. e. keine eingehenden Kanten), werden in eine Liste S gepackt. Alle Knoten, welche auf diese Knoten verweisen, werden "gelöst" von ebendiesen. Danach werden die Knoten aus S in ein weiteres Feld L eingetragen. Dieses Feld wird am schluss die geordnete Menge an Knoten enthalten. Nun werden die zuvor gelösten Knoten in S eingeschrieben. Der ganze Prozess wird dann so lange wiederholt, bis im Graphen keine Kanten mehr verzeichnet sind. Gibt es doch noch welche, ist der Graph zyklisch und kann nicht topologisch sortiert werden. (Quelle: Wikipedia)

2.2 Implementierung

Listing 9: *main.c*

```c
#include <stdio.h>
#include <stdbool.h>
#include "directed_graph.h"
#include "graph_types.h"

#define MAX 100

// Sorting a graph using kahns algorithm
void sort_topologically(graph_ptr pgraph, conscious_node nodes[], bool *error
    )
{
    /*
     * In order do differentiate between my comments and the
     * outline of kahns algorithm, I put the outline comments
     * at the beginning of a line.
     */

    size_t node_count = nr_of_nodes(pgraph);
    if (error != NULL) {
        *error = false;
    }

//      L Empty list that will contain the sorted elements
    conscious_node *L = (conscious_node*) malloc(sizeof(conscious_node) *
     node_count);
    size_t li = 0;

//      S Set of all nodes with no incoming edge
    conscious_node *S = (conscious_node*) malloc(sizeof(conscious_node) *
     node_count);
    size_t si = 0;

    conscious_node *All = get_conscious_nodes(pgraph);

    // move all nodes without dependencies to S
```

```
33      for (size_t i = 0; i < node_count; ++i) {
34          if (All[i].nr_of_dependencies == 0) {
35              S[si++] = All[i];
36          }
37      }
38
39  //    while S is non-empty do
40      while (si > 0) {
41  //          remove a node n from S
42          conscious_node n = S[--si];
43  //          add n to tail of L
44          L[li++] = n;
45  //          for each node m with an edge e from n to m do
46          for (size_t potential_m_index = 0; potential_m_index < node_count; ++
        potential_m_index) {
47              conscious_node potential_m = All[potential_m_index];
48              size_t m_dep_index = 0;
49              while (m_dep_index < potential_m.nr_of_dependencies &&
        potential_m.dependency_arr[m_dep_index] != n.data.id) {
50                  ++m_dep_index;
51              }
52              if (m_dep_index < potential_m.nr_of_dependencies) {
53                  conscious_node m = potential_m;
54  //                remove edge e from the graph
55                  remove_graph_edge(pgraph, n.data.id, m.data.id);
56  //                if m has no other incoming edges then
57                  if (node_in_degree(pgraph, m.data.id) == 0) {
58  //                  insert m into S
59                      S[si++] = m;
60                  }
61              }
62          }
63      }
64
65      if (nr_of_edges(pgraph) != 0 && error != NULL) {
66          *error = true;
67      }
68      for (size_t cpy = 0; cpy < node_count; ++cpy) {
69          nodes[cpy] = L[cpy];
70          free(All[cpy].dependency_arr);
71      }
72
73      free(All);
74      free(L);
75      free(S);
76  }
77
78  void print_sort_result(const cgraph_ptr cpgraph, const conscious_node *nodes,
        bool error)
79  {
80      if (error) {
81          printf("Graph was cyclic");
82      } else {
83          for (size_t i = 0; i < nr_of_nodes(cpgraph); ++i) {
84              printf("( %s ), ", nodes[i].data.payload);
85          }
86          printf("\n");
```

```
87      }
88  }
89
90  int main()
91  {
92      conscious_node nodes[MAX];
93      bool error = false;
94      graph_ptr pgraph = create_graph();
95
96      // construct graph
97      node_id work = add_graph_node(pgraph, "working out a solution");
98      node_id finish = add_graph_node(pgraph, "finishing");
99      node_id read = add_graph_node(pgraph, "reading the requirements");
100     add_graph_edge(pgraph, read, work);
101     add_graph_edge(pgraph, work, finish);
102
103     sort_topologically(pgraph, nodes, &error);
104     print_sort_result(pgraph, nodes, error);
105
106     // construct cyclic graph
107     graph_ptr cyclic_graph = create_graph();
108     node_id a = add_graph_node(cyclic_graph, "A");
109     add_graph_edge(cyclic_graph, a, a);
110
111     sort_topologically(cyclic_graph, nodes, &error);
112     print_sort_result(cyclic_graph, nodes, error);
113
114     delete_graph(&pgraph);
115     delete_graph(&cyclic_graph);
116     return 0;
117 }
```

2.3 Tests

```
/home/niklas/Documents/Github/fh-hgb-ws1819/swo/ue05/cmake-build-debug/top
( reading the requirements ), ( working out a solution ), ( finishing ),
Graph was cyclic
Process finished with exit code 0
```

Figure 2: Resultat von *top/main.c*