SWO 3x Übung zu Softwareentwicklung mit klassischen Sprachen u. Bibliotheken 3 **WS 2014/15, Übung 03**

Abgabetermin: Sa in der KW 43

☐ **Gr. 1,** DI Franz Gruber-Leitner **Name** Roman Lumetsberger **Aufwand in h** 7

☒ **Gr. 2,** Dr. Erik Pitzer

Punkte _____ **Kurzzeichen Tutor / Übungsleiter** _____ / _____

1. Repräsentation von *gewichteten* Graphen (7 + 9 Punkte)

Hinweis: Es werden nur Graphen mit einer fixen Anzahl von *n* Knoten betrachtet, d.h. die Anzahl der Knoten kann nicht wachsen oder schrumpfen und muss von Anfang an bekannt sein. Als "Knotennamen" können deshalb die ganzzahligen Werte von (z.B.) 1 bis *n* dienen.

a) Entwickeln Sie einen *abstrakten Datentyp* (in Form eines Moduls *DG_ADT_**M**.c*, wobei *DG* für *directed graph* und *M* für *matrix* steht) zur Verwaltung eines gewichteten Graphen mit *n* Knoten, wobei intern zur Repräsentation eine *Adjazenz**m**atrix* verwendet wird.

b) Entwickeln Sie einen *abstrakten Datentyp* (in Form eines Moduls *DG_ADT_**L**.c*, wobei *L* für *list* steht) zur Verwaltung eines gewichteten Graphen mit *n* Knoten, wobei zur Repräsentation intern eine *Adjazenz**l**iste* verwendet wird.

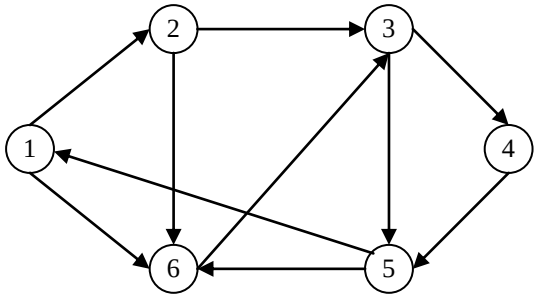
Die Mindestfunktionalität umfasst in beiden Fällen

1. das Erzeugen eines neuen Graphen mit dem Parameter *n*,
2. das Eintragen neuer Kanten zwischen Knoten mit Gewicht *w*,
3. das Löschen von Kanten zwischen Knoten,
4. das Ausgeben der internen Darstellung des Graphen in geeigneter Form und
5. das Freigeben eines Graphen.

Achten Sie darauf, dass beide Datenstrukturen "austauschbar" sind – indem beide Implementierungen die gleiche Schnittstelle *DG_ADT.h* erfüllen und durch binden („linken") die eine oder die andere Implementierung eingebunden werden kann.

Schreiben Sie *ein* Testprogramm *für beide* Datentypen, in dem Sie den Graphen aus der Abbildung rechts aufbauen und die interne Darstellung ausgeben. Sie können dabei beliebige positive Kantengewichte annehmen.

Testen Sie auch die Möglichkeit, eine Kante zu löschen.



So könnte die Schnittstelle in DG_ADT.h anfangs aussehen. Ein wichtiges Detail dabei ist, dass das „`struct graph`" nur deklariert, jedoch nicht definiert wird, was Sie dann in der konkreten Implementierung erst machen können.

```
struct Graph;
typedef struct Graph Graph;

Graph* createGraph(int n);
void freeGraph(Graph *g);
void insertEdge(Graph *g, int source, int target, double weight);
void removeEdge(Graph *g, int source, int target);
void printGraph(Graph *g);
```

2. Graphenalgorithmen (1 + 3 + 4 Punkte)

Erweitern Sie die Schnittstelle in DG_ADT.H sowie beide Implementierungen nach eigenem Ermessen um die Realisierung der nun folgenden weiteren Funktionalität **implementierungs-unabhängig** zu ermöglichen, also nur abhängig von der allgemeinen Schnittstelle DG_ADT.H. Halten Sie dazu diese Schnittstelle möglichst klein und allgemein.

Erstellen Sie zwei neue Dateien GRAPH_ALGS.H und GRAPH_ALGS.C in der Sie folgende Funktionen zur Verfügung stellen:

1. Erzeugung des invertierten Graphen (Alle Kanten werden umgedreht)
2. Testen auf Reflexivität, Symmetrie, Asymmetrie und Transitivität eines Graphen

Die Schnittstelle dieser Algorithmen könnte z.B. so aussehen:

```
#include "dg_adt.h"

Graph* invert(Graph *g);
bool isReflexive(Graph *g);
bool isSymmetric(Graph *g);
bool isAsymmetric(Graph *g);
bool isTransitive(Graph *g);
void printGraphProperties(Graph *g);
```

Hinweise:

1. Geben Sie für alle Ihre Lösungen immer eine Lösungsidee an.
2. Kommentieren und testen Sie Ihre Programme ausführlich.

1 Aufgabe 1 - Repräsentation von gewichteten Graphen

1.1 Lösungsidee

Zu Beginn muss das Interface *dg_adt.h* definiert werden. Dabei wird die Schnittstelle der Angabe verwendet. Wichtig ist hier, dass der Type *graph* deklariert wird, ohne eine konkrete interne Struktur vorzugeben.
In diesem Beispiel werden als Knotennamen 1 - n verwendet, wobei intern die Speicherung mit Index 0 erfolgt.

1.1.1 Adjazenzmatrix

Beim diesem Beispiel wird der Graph als Adjazenzmatrix implementiert.
Eine Adjazenzmatrix ist eine Matrix mit n * n Elementen. Dabei wird für jede mögliche Kombination das Gewicht der Kante gespeichert.
Ist das Gewicht Null, dann existiert keine Kante zwischen den Knoten.

- Beim Einfügen einer Kante wird einfach das Gewicht an die richtige Stelle der Matrix geschrieben.
- Beim Löschen wird der Wert wieder auf Null gesetzt.

Speicherverwaltung
Beim Erstellen des Graphen wird die gesamte Matrix allokiert, beim Löschen wird sie wieder freigegeben.

1.1.2 Adjazenzliste

Bei dieser Variante der Speicherung eines Graphen wird ein Array verwendet, welches für jeden Knoten eine Liste der Kanten bereitstellt.
Die Implementierung verwendet eine einfach verkette Liste.

- Beim Einfügen einer Kante wird ein neues Element in die Liste des Ausgangsknoten eingefügt
- Beim Löschen wird das Element der Liste wieder entfernt

Speicherverwaltung
Beim Erstellen des Graphen wird nur das Array mit den Zeigern auf die Liste allokiert.

- Wird ein Element eingefügt, wird ein neues Listenelement allokiert und in die Liste eingefügt.
- Beim Löschen einer Kante muss das Element gesucht, die Verkettung der Liste aktualisiert und das Element freigegeben werden.
- Beim Löschen des Graphen müssen zuerst die Listenelemente, dann das Array der Knoten und dann der Graph selbst freigegeben werden.

1.2 Sourcecode

1.2.1 Makefile

In der Abgabe Zip Datei ist ein Makefile enthalten, dass für jede Implementierung ein Target enthält.

- Target graphm: Erstellt das Programm **graphm**, das als Implementierung die Adjazenzmatrix (dg_adt_m.o) verwendet
- Target graphl: Erstellt das Programm **graphl**, das als Implementierung die Adjazenzliste (dg_adt_l.o) verwendet

```c
/*****************************************************************************
  dg_adt.h
  Roman Lumetsberger

  defines the interface for graphs
*****************************************************************************/
#ifndef DG_ADT_INCLUDED
#define DG_ADT_INCLUDED

struct Graph;
typedef struct Graph Graph;

/* Creates a graph with n vertices */
Graph* createGraph(int n);
/* Frees the graph memory */
void freeGraph(Graph *g);
/* inserts a edge between the source and target
   source and target starts with 1 */
void insertEdge(Graph *g, int source, int target, double weight);
/* removes the edge between the source and target
   source and target start with 1 */
void removeEdge(Graph *g, int source, int target);
/* prints the graph */
void printGraph(Graph *g);

#endif
```

```c
/*****************************************************************************
  dg_adt_m.c
  Roman Lumetsberger

  Implements the graph interface as matrix
*****************************************************************************/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

```c
#include <assert.h>
#include "dg_adt.h"

struct Graph {
  /* count of vertices */
  int verticesCount;
  /* matrix */
  double *matrix;
};

/* Create the graph with the given vertex count*/
Graph* createGraph(int n) {
  Graph *graph =(Graph*) malloc(sizeof(Graph));

  /* return null if there is not have enough memory */
  if (graph == NULL)
    return NULL;

  graph->verticesCount = n;
  /* reserve the memory for the matrix */
  graph->matrix = (double *) malloc(sizeof(double) * n * n);

  /* free the graph and return null if there is not enough memory */
  if (graph->matrix == NULL) {
    free(graph);
    return NULL;
  }
  /* set the matrix to 0 */
  memset(graph->matrix, 0, sizeof(double) * n * n);
  return graph;
}

/* frees the graph */
void freeGraph(Graph *g) {
  assert( g != NULL);
  free(g->matrix);
  free(g);
}

/* inserts a edge in the matrix
   source and target starts with 1 and are saved as 0 */
void insertEdge(Graph *g, int source, int target, double weight) {
  assert (g != NULL);
  assert (source > 0 && source <= g->verticesCount);
  assert (target > 0 && target <= g->verticesCount);
  g->matrix[((source - 1) * g->verticesCount) + target - 1] = weight;
}

/* removes a edge in the matrix
```

```
59    source and target starts with 1 and are saved as 0 */
60 void removeEdge(Graph *g, int source, int target) {
61   assert (g != NULL);
62   assert (source > 0 && source <= g->verticesCount);
63   assert (target > 0 && target <= g->verticesCount);
64   g->matrix[((source - 1) * g->verticesCount) + target - 1] = 0;
65 
66 }
67 
68 /* prints the graph matrix */
69 void printGraph(Graph *g) {
70   int i, j;
71   printf("     ");
72   for (i = 0; i < g->verticesCount; i++) {
73     printf("   %d ", i + 1);
74   }
75   printf("\n");
76   for (i = 0; i < g->verticesCount; i++) {
77     printf("-----");
78   }
79   printf("-----\n");
80   for (i = 0; i < g->verticesCount; i++) {
81     for (j = 0; j < g->verticesCount; j++) {
82       if (j == 0) {
83         printf(" %d | ", i + 1);
84       }
85       printf("%4.2g ",g->matrix[i * g->verticesCount +j]);
86     }
87     printf("\n");
88   }
89 }
```

```
1  /*****************************************************************************
2    dg_adt_l.c
3    Roman Lumetsberger
4 
5    Implements the graph interface as list
6  *****************************************************************************/
7  #include <stdlib.h>
8  #include <stdio.h>
9  #include <string.h>
10 #include <assert.h>
11 #include "dg_adt.h"
12 
13 /* defines the internal vertex list */
14 typedef struct VertexList {
15   int target;
16   double weight;
```

```
17   struct VertexList *next;
18 } VertexList;
19
20 struct Graph {
21   /* count of vertices */
22   int verticesCount;
23   VertexList **adjList;
24 };
25
26 /* Create the graph with the given vertex count */
27 Graph* createGraph(int n) {
28   Graph *graph =(Graph*) malloc(sizeof(Graph));
29
30   /* return null if there is not have enough memory */
31   if (graph == NULL)
32     return NULL;
33
34   graph->verticesCount = n;
35   /* reserve the memory for the list */
36   graph->adjList = (VertexList **) malloc(sizeof(VertexList*) * n);
37
38   /* free the graph and return null if there is not enough memory */
39   if (graph->adjList == NULL) {
40     free(graph);
41     return NULL;
42   }
43   /* set the vertex list to 0 */
44   memset(graph->adjList, 0, sizeof(VertexList*) * n);
45   return graph;
46 }
47
48 /* frees the graph */
49 void freeGraph(Graph *g) {
50   int i;
51   VertexList *current, *next;
52
53   assert( g != NULL);
54
55   /* frees the list elements */
56   for (i = 0; i < g->verticesCount; i++) {
57     current = g->adjList[i];
58     while (current != NULL) {
59       next = current->next;
60       free (current);
61       current = next;
62     }
63   }
64   /* frees the list array */
65   free (g->adjList);
```

```
66    /* frees the graph */
67    free(g);
68  }
69
70  /* inserts a edge
71     source and target start with 1 */
72  void insertEdge(Graph *g, int source, int target, double weight) {
73    VertexList *curElement;
74    assert (g != NULL);
75    assert (source > 0 && source <= g->verticesCount);
76    assert (target > 0 && target <= g->verticesCount);
77    curElement = g->adjList[source - 1];
78
79    /* try to find the element */
80    while (curElement != NULL && curElement->target != (target - 1)) {
81      curElement = curElement->next;
82    }
83
84    /* element found, change the weight */
85    if (curElement != NULL) {
86      curElement->weight = weight;
87      return;
88    }
89
90    curElement = (VertexList *)malloc(sizeof(VertexList));
91    if(curElement == NULL) {
92      printf("Out of memory\n");
93      exit(-1);
94    }
95
96    curElement->target = target - 1;
97    curElement->weight = weight;
98
99    /* first edge */
100   if (g->adjList[source - 1] == NULL) {
101     g->adjList[source - 1] = curElement;
102     curElement->next = NULL;
103   }
104   else {
105     curElement->next = g->adjList[source - 1];
106     g->adjList[source - 1] = curElement;
107   }
108 }
109
110 /* removes a edge in the list
111    source and target start with 1 */
112 void removeEdge(Graph *g, int source, int target) {
113   VertexList *curElement;
114   VertexList *parent;
```

```
115
116   assert (g != NULL);
117   assert (source > 0 && source <= g->verticesCount);
118   assert (target > 0 && target <= g->verticesCount);
119
120   curElement = g->adjList[source - 1];
121   parent = NULL;
122   while (curElement != NULL && curElement->target != (target - 1)) {
123     parent = curElement;
124     curElement = curElement->next;
125   }
126
127   /* Element found */
128   if (curElement != NULL ) {
129     if (parent == NULL) {
130       g->adjList[source - 1] = curElement->next;
131       free(curElement);
132     }
133     else {
134       parent->next = curElement->next;
135       free(curElement);
136     }
137   }
138
139 }
140
141 /* prints the graph list */
142 void printGraph(Graph *g) {
143   int i;
144   VertexList *curElement;
145
146   assert (g != NULL);
147   for (i = 0; i < g->verticesCount; i++) {
148     curElement = g->adjList[i];
149
150     /* vertex names start with 1 */
151     printf("%d ",i + 1);
152     while (curElement != NULL) {
153       printf("--> %d(%4.2g) ",curElement->target + 1, curElement->weight);
154       curElement = curElement->next;
155     }
156     printf("\n");
157   }
158
159 }
```

```
1 /*****************************************************************************
2   graph.c
```

```
3   Roman Lumetsberger
4
5   Testprogram for graph functions without using concrete implementation
6 ******************************************************************************/
7 #include <stdlib.h>
8 #include <stdio.h>
9 #include "dg_adt.h"
10
11 int main() {
12
13   Graph *g = createGraph(6);
14   printf("Empty graph:\n");
15   printGraph(g);
16   insertEdge(g, 1, 2, 2.3);
17   insertEdge(g, 1, 6, 1.6);
18   /*overwrite edge */
19   insertEdge(g, 1, 6, 1.3);
20
21   insertEdge(g, 2, 6, 0.3);
22   insertEdge(g, 2, 3, 1.5);
23   insertEdge(g, 3, 5, 6.4);
24   insertEdge(g, 3, 4, 3.4);
25   insertEdge(g, 4, 5, 1.3);
26   insertEdge(g, 5, 1, 2);
27   insertEdge(g, 6, 3, 1);
28
29   printf("\n\nTest graph:\n");
30   printGraph(g);
31
32   removeEdge(g, 1, 6);
33   removeEdge(g, 1, 2);
34   /* remove edge which does not exist */
35   removeEdge(g, 1, 3);
36   removeEdge(g, 2, 6);
37   removeEdge(g, 3, 4);
38   removeEdge(g, 3, 4);
39   printf("\n\nRemoved edge graph:\n");
40   printGraph(g);
41   freeGraph(g);
42   /* set g to NULL, because pointer is not valid anymore */
43   g = NULL;
44   return EXIT_SUCCESS;
45 }
```

1.3 Testfälle

1.3.1 Testfall 1 - Adjazenzmatrix

- Erzeugen eines Graphen mit 6 Knoten

- Ausgeben des leeren Graphen
- Hinzufügen aller Kanten des Angabenbeispiels
- Überschreiben einer Kante mit einem anderen Gewicht
- Ausgeben des Graphen
- Löschen einiger Kanten
- Ausgeben des Graphen
- Freigeben des Graphen

```
romanlum@ubuntu: ~/swo3/UebungMoodle3/Beispiel1
romanlum@ubuntu:~/swo3/UebungMoodle3/Beispiel1$ ./graphm
Empty graph:
        1    2    3    4    5    6
---------------------------------
 1 |    0    0    0    0    0    0
 2 |    0    0    0    0    0    0
 3 |    0    0    0    0    0    0
 4 |    0    0    0    0    0    0
 5 |    0    0    0    0    0    0
 6 |    0    0    0    0    0    0


Test graph:
        1    2    3    4    5    6
---------------------------------
 1 |    0  2.3    0    0    0  1.3
 2 |    0    0  1.5    0    0  0.3
 3 |    0    0    0  3.4  6.4    0
 4 |    0    0    0    0  1.3    0
 5 |    2    0    0    0    0    0
 6 |    0    0    1    0    0    0


Removed edge graph:
        1    2    3    4    5    6
---------------------------------
 1 |    0    0    0    0    0    0
 2 |    0    0  1.5    0    0    0
 3 |    0    0    0    0  6.4    0
 4 |    0    0    0    0  1.3    0
 5 |    2    0    0    0    0    0
 6 |    0    0    1    0    0    0
romanlum@ubuntu:~/swo3/UebungMoodle3/Beispiel1$
```

1.3.2 Testfall 2 - Adjazenzmatrix - Valgrind

Gleiche Operationen wie in Testfall 1, nur diesmal mit valgrind (Memory Leak Detector) ausgeführt, um zu zeigen, dass der Speicher wieder korrekt freigeben wird.

```
romanlum@ubuntu: ~/swo3/UebungMoodle3/Beispiel1
romanlum@ubuntu:~/swo3/UebungMoodle3/Beispiel1$ valgrind ./graphm
==4964== Memcheck, a memory error detector
==4964== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==4964== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==4964== Command: ./graphm
==4964==
Empty graph:
        1    2    3    4    5    6
-----------------------------------
 1 |    0    0    0    0    0    0
 2 |    0    0    0    0    0    0
 3 |    0    0    0    0    0    0
 4 |    0    0    0    0    0    0
 5 |    0    0    0    0    0    0
 6 |    0    0    0    0    0    0


Test graph:
        1    2    3    4    5    6
-----------------------------------
 1 |    0  2.3    0    0    0  1.3
 2 |    0    0  1.5    0    0  0.3
 3 |    0    0    0  3.4  6.4    0
 4 |    0    0    0    0  1.3    0
 5 |    2    0    0    0    0    0
 6 |    0    0    1    0    0    0


Removed edge graph:
        1    2    3    4    5    6
-----------------------------------
 1 |    0    0    0    0    0    0
 2 |    0    0  1.5    0    0    0
 3 |    0    0    0    0  6.4    0
 4 |    0    0    0    0  1.3    0
 5 |    2    0    0    0    0    0
 6 |    0    0    1    0    0    0
==4964==
==4964== HEAP SUMMARY:
==4964==     in use at exit: 0 bytes in 0 blocks
==4964==   total heap usage: 2 allocs, 2 frees, 296 bytes allocated
==4964==
==4964== All heap blocks were freed -- no leaks are possible
==4964==
==4964== For counts of detected and suppressed errors, rerun with: -v
==4964== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
romanlum@ubuntu:~/swo3/UebungMoodle3/Beispiel1$
```

1.3.3 Testfall 3 - Adjazenzliste

Gleiche Operationen wie in Testfall 1, nur diesmal mit der Listenimplementierung

```
romanlum@ubuntu: ~/swo3/UebungMoodle3/Beispiel1
romanlum@ubuntu:~/swo3/UebungMoodle3/Beispiel1$ ./graphl
Empty graph:
1
2
3
4
5
6


Test graph:
1 --> 6( 1.3) --> 2( 2.3)
2 --> 3( 1.5) --> 6( 0.3)
3 --> 4( 3.4) --> 5( 6.4)
4 --> 5( 1.3)
5 --> 1(   2)
6 --> 3(   1)


Removed edge graph:
1
2 --> 3( 1.5)
3 --> 5( 6.4)
4 --> 5( 1.3)
5 --> 1(   2)
6 --> 3(   1)
romanlum@ubuntu:~/swo3/UebungMoodle3/Beispiel1$
```

1.3.4 Testfall 4 - Adjazenzliste - Valgrind

Gleiche Operationen wie in Testfall 3, nur diesmal mit valgrind (Memory Leak Detector) ausgeführt, um zu zeigen, dass der Speicher wieder korrekt freigeben wird.

```
romanlum@ubuntu: ~/swo3/UebungMoodle3/Beispiel1
romanlum@ubuntu:~/swo3/UebungMoodle3/Beispiel1$ valgrind ./graphl
==3945== Memcheck, a memory error detector
==3945== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==3945== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==3945== Command: ./graphl
==3945==
Empty graph:
1
2
3
4
5
6


Test graph:
1 --> 6( 1.3) --> 2( 2.3)
2 --> 3( 1.5) --> 6( 0.3)
3 --> 4( 3.4) --> 5( 6.4)
4 --> 5( 1.3)
5 --> 1(   2)
6 --> 3(   1)


Removed edge graph:
1
2 --> 3( 1.5)
3 --> 5( 6.4)
4 --> 5( 1.3)
5 --> 1(   2)
6 --> 3(   1)
==3945==
==3945== HEAP SUMMARY:
==3945==     in use at exit: 0 bytes in 0 blocks
==3945==   total heap usage: 11 allocs, 11 frees, 176 bytes allocated
==3945==
==3945== All heap blocks were freed -- no leaks are possible
==3945==
==3945== For counts of detected and suppressed errors, rerun with: -v
==3945== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
romanlum@ubuntu:~/swo3/UebungMoodle3/Beispiel1$
```

2 Aufgabe 2 - Graphenalgorithmen

2.1 Lösungsidee

Um die benötigten Algorithmen umzusetzen werden 2 neue Methoden benötigt

```c
/* gets the count of vertices */
int getVerticesCount(Graph *g);

/* gets the weight of the edge between source and target
returns 0 if there is no edge
source and target are starting with 1 */
double getEdgeWeight(Graph *g, int source, int target);
```

Mit diesen Methoden können dann in einer Schleife alle Knoten durchlaufen und die Kanten geprüft werden.

2.1.1 invert

Um den Graphen zu invertieren, muss zuerst ein neuer Graph mit der korrekten Anzahl an Knoten angelegt werden.
Dann müssen alle Knoten in einer verschachtelten Schleife durchlaufen werden

- Gibt es eine Kante muss in den neuen Graph eine Kante in umgekehrter Richtung eingefügt werden.

2.1.2 isReflexive

Ein Graph ist reflexiv, wenn jeder Knoten eine Kante auf sich selbst hat.
D.h. alle Knoten durchlaufen und prüfen, ob eine Kante zu sich selbst existiert.

2.1.3 isSymmetric

Ein Graph ist symmetrisch, wenn jede Kante in beiden Richtungen vorhanden ist.
D.h. alle Kanten durchlaufen und prüfen, ob auch die Kante in umgekehrter Richtung existiert.

2.1.4 isAsymmetric

Ein Graph ist asymmetrisch, wenn jede Kante nur in einer Richtungen vorhanden ist.
D.h. alle Kanten durchlaufen und prüfen, dass keine Kante in umgekehrter Richtung existiert.

2.1.5 isTransitivev

Ein Graph ist transitiv, wenn jeder Knoten, der über einen anderen Knoten erreichbar ist, auch eine direkte Kante hat.
D.h. alle Knoten und deren Pfade durchlaufen und prüfen ob auch eine direkte Verbindung vom Ausgangsknoten existiert.

2.1.6 printGraphProperties

Wendet die oben angeführten Funktionen auf den Graph an und gibt die Ergebnisse aus.

2.2 Sourcecode

2.2.1 Makefile

In der Abgabe Zip Datei ist ein Makefile enthalten, dass für jede Implementierung ein Target enthält.

- Target graphm: Erstellt das Programm **graphm**, das als Implementierung die Adjazenzmatrix (dg_adt_m.o) verwendet
- Target graphl: Erstellt das Programm **graphl**, das als Implementierung die Adjazenzliste (dg_adt_l.o) verwendet

```c
/*****************************************************************************
  dg_adt.h
  Roman Lumetsberger

  defines the interface for graphs
*****************************************************************************/
#ifndef DG_ADT_INCLUDED
#define DG_ADT_INCLUDED

struct Graph;
typedef struct Graph Graph;

/* Creates a graph with n vertices */
Graph* createGraph(int n);
/* Frees the graph memory */
void freeGraph(Graph *g);
/* inserts a edge between the source and target
   source and target starts with 1 */
void insertEdge(Graph *g, int source, int target, double weight);
/* removes the edge between the source and target
   source and target start with 1 */
void removeEdge(Graph *g, int source, int target);
/* prints the graph */
void printGraph(Graph *g);

/* gets the count of vertices */
int getVerticesCount(Graph *g);

/* gets the weight of the edge between source and target
   returns 0 if there is no edge
   source and target are starting with 1 */
double getEdgeWeight(Graph *g, int source, int target);

#endif
```

```c
/*****************************************************************************
  dg_adt_m.c
  Roman Lumetsberger

  Implements the graph interface as matrix
*****************************************************************************/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include "dg_adt.h"

```

```c
struct Graph {
  /* count of vertices */
  int verticesCount;
  /* matrix */
  double *matrix;
};

/* Create the graph with the given vertex count*/
Graph* createGraph(int n) {
  Graph *graph =(Graph*) malloc(sizeof(Graph));

  /* return null if there is not have enough memory */
  if (graph == NULL)
    return NULL;

  graph->verticesCount = n;
  /* reserve the memory for the matrix */
  graph->matrix = (double *) malloc(sizeof(double) * n * n);

  /* free the graph and return null if there is not enough memory */
  if (graph->matrix == NULL) {
    free(graph);
    return NULL;
  }
  /* set the matrix to 0 */
  memset(graph->matrix, 0, sizeof(double) * n * n);
  return graph;
}

/* frees the graph */
void freeGraph(Graph *g) {
  assert( g != NULL);
  free(g->matrix);
  free(g);
}

/* inserts a edge in the matrix
   source and target starts with 1 and are saved as 0 */
void insertEdge(Graph *g, int source, int target, double weight) {
  assert (g != NULL);
  assert (source > 0 && source <= g->verticesCount);
  assert (target > 0 && target <= g->verticesCount);
  g->matrix[((source - 1) * g->verticesCount) + target - 1] = weight;
}

/* removes a edge in the matrix
   source and target starts with 1 and are saved as 0 */
void removeEdge(Graph *g, int source, int target) {
  assert (g != NULL);
```

```
62    assert (source > 0 && source <= g->verticesCount);
63    assert (target > 0 && target <= g->verticesCount);
64    g->matrix[((source - 1) * g->verticesCount) + target - 1] = 0;
65
66  }
67
68  /* prints the graph matrix */
69  void printGraph(Graph *g) {
70    int i, j;
71    printf("     ");
72    for (i = 0; i < g->verticesCount; i++) {
73      printf("   %d ", i + 1);
74    }
75    printf("\n");
76    for (i = 0; i < g->verticesCount; i++) {
77      printf("-----");
78    }
79    printf("-----\n");
80    for (i = 0; i < g->verticesCount; i++) {
81      for (j = 0; j < g->verticesCount; j++) {
82        if (j == 0) {
83          printf(" %d | ", i + 1);
84        }
85        printf("%4.2g ",g->matrix[i * g->verticesCount +j]);
86      }
87      printf("\n");
88    }
89  }
90
91  /* returns the vertices count of the given graph */
92  int getVerticesCount(Graph *g) {
93    assert (g != NULL);
94    return g->verticesCount;
95  }
96
97  /* gets the weight of the edge between source and target
98     return 0 if there is no edge
99     source and target are starting with 1 */
100 double getEdgeWeight(Graph *g, int source, int target) {
101   assert (g != NULL);
102   assert (source > 0 && source <= g->verticesCount);
103   assert (target > 0 && target <= g->verticesCount);
104   return g->matrix[((source - 1)*g->verticesCount) + (target - 1)];
105 }
```

```
1  /*****************************************************************************
2    dg_adt_l.c
3    Roman Lumetsberger
```

```c

  Implements the graph interface as list
*****************************************************************************/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include "dg_adt.h"

/* defines the internal vertex list */
typedef struct VertexList {
  int target;
  double weight;
  struct VertexList *next;
} VertexList;

struct Graph {
  /* count of vertices */
  int verticesCount;
  VertexList **adjList;
};

/* Create the graph with the given vertex count */
Graph* createGraph(int n) {
  Graph *graph =(Graph*) malloc(sizeof(Graph));

  /* return null if there is not have enough memory */
  if (graph == NULL)
    return NULL;

  graph->verticesCount = n;
  /* reserve the memory for the list */
  graph->adjList = (VertexList **) malloc(sizeof(VertexList*) * n);

  /* free the graph and return null if there is not enough memory */
  if (graph->adjList == NULL) {
    free(graph);
    return NULL;
  }
  /* set the vertex list to 0 */
  memset(graph->adjList, 0, sizeof(VertexList*) * n);
  return graph;
}

/* frees the graph */
void freeGraph(Graph *g) {
  int i;
  VertexList *current, *next;

```

```
53    assert( g != NULL);
54
55    /* frees the list elements */
56    for (i = 0; i < g->verticesCount; i++) {
57      current = g->adjList[i];
58      while (current != NULL) {
59        next = current->next;
60        free (current);
61        current = next;
62      }
63    }
64    /* frees the list array */
65    free (g->adjList);
66    /* frees the graph */
67    free(g);
68  }
69
70  /* inserts a edge
71     source and target start with 1 */
72  /* inserts a edge
73     source and target start with 1 */
74  void insertEdge(Graph *g, int source, int target, double weight) {
75    VertexList *curElement;
76    assert (g != NULL);
77    assert (source > 0 && source <= g->verticesCount);
78    assert (target > 0 && target <= g->verticesCount);
79
80    curElement = g->adjList[source - 1];
81
82    /* try to find the element */
83    while (curElement != NULL && curElement->target != (target - 1)) {
84      curElement = curElement->next;
85    }
86
87    /* element found, change the weight */
88    if (curElement != NULL) {
89      curElement->weight = weight;
90      return;
91    }
92
93    curElement = (VertexList *)malloc(sizeof(VertexList));
94    if(curElement == NULL) {
95      printf("Out of memory\n");
96      exit(-1);
97    }
98
99    curElement->target = target - 1;
100   curElement->weight = weight;
101
```

```
102   /* first edge */
103   if (g->adjList[source - 1] == NULL) {
104     g->adjList[source - 1] = curElement;
105     curElement->next = NULL;
106   }
107   else {
108     curElement->next = g->adjList[source - 1];
109     g->adjList[source - 1] = curElement;
110   }
111 }
112
113
114 /* removes a edge in the list
115    source and target start with 1 */
116 void removeEdge(Graph *g, int source, int target) {
117   VertexList *curElement;
118   VertexList *parent;
119
120   assert (g != NULL);
121   assert (source > 0 && source <= g->verticesCount);
122   assert (target > 0 && target <= g->verticesCount);
123
124   curElement = g->adjList[source - 1];
125   parent = NULL;
126   while (curElement != NULL && curElement->target != (target - 1)) {
127     parent = curElement;
128     curElement = curElement->next;
129   }
130
131   /* Element found */
132   if (curElement != NULL ) {
133     if (parent == NULL) {
134       g->adjList[source - 1] = curElement->next;
135       free(curElement);
136     }
137     else {
138       parent->next = curElement->next;
139       free(curElement);
140     }
141   }
142
143 }
144
145 /* prints the graph list */
146 void printGraph(Graph *g) {
147   int i;
148   VertexList *curElement;
149
150   assert (g != NULL);
```

```
151   for (i = 0; i < g->verticesCount; i++) {
152     curElement = g->adjList[i];
153 
154     /* vertex names start with 1 */
155     printf("%d ",i + 1);
156     while (curElement != NULL) {
157       printf("--> %d(%4.2g) ",curElement->target + 1, curElement->weight);
158       curElement = curElement->next;
159     }
160     printf("\n");
161   }
162 
163 }
164 
165 
166 /* returns the vertices count of the given graph */
167 int getVerticesCount(Graph *g) {
168   assert (g != NULL);
169   return g->verticesCount;
170 }
171 
172 /* gets the weight of the edge between source and target
173    return 0 if there is no edge
174    source and target are starting with 1 */
175 double getEdgeWeight(Graph *g, int source, int target) {
176   VertexList *curElement;
177 
178   assert (g != NULL);
179   assert (source > 0 && source <= g->verticesCount);
180   assert (target > 0 && target <= g->verticesCount);
181 
182   curElement = g->adjList[source - 1];
183 
184   while (curElement != NULL) {
185     if (curElement->target == target - 1)
186       return curElement->weight;
187 
188     curElement = curElement->next;
189   }
190 
191   return 0;
192 }
```

```
1 /*****************************************************************************
2   graph_algs.h
3   Roman Lumetsberger
4 
5   defines the interface for graph functions
```

```
6  ********************************************************************************/
7
8  #ifndef GRAPH_ALGS_INCLUDED
9  #define GRAPH_ALGS_INCLUDED
10
11 #include "dg_adt.h"
12
13 typedef enum bool {false, true} bool;
14
15 /* creates a new inverted graph */
16 Graph* invert(Graph *g);
17 /* checks if the graph is reflexiv */
18 bool isReflexive(Graph *g);
19 /* checks if the graph is symmetrix */
20 bool isSymmetric(Graph *g);
21 /* checks if the graph is asymmetrix */
22 bool isAsymmetric(Graph *g);
23 /* checks if the graph is transitiv */
24 bool isTransitive(Graph *g);
25 /* prints all the graph properties */
26 void printGraphProperties(Graph *g);
27
28 #endif
```

```
1  /********************************************************************************
2    graph_algs.c
3    Roman Lumetsberger
4
5    implementation for the graph functions
6  ********************************************************************************/
7  #include <stdlib.h>
8  #include <string.h>
9  #include <stdio.h>
10 #include <assert.h>
11 #include "graph_algs.h"
12
13 /* creates a new inverted graph */
14 Graph* invert(Graph *g) {
15   int edgeCount;
16   int i,j;
17   double weight;
18   Graph *newGraph;
19
20   assert (g != NULL);
21   edgeCount = getVerticesCount(g);
22   newGraph = createGraph(edgeCount);
23   if(newGraph == NULL)
24     return NULL;
```

```
25
26    for (i = 1; i <= edgeCount; i++) {
27      for (j = 1; j <= edgeCount; j++) {
28        weight = getEdgeWeight(g, i, j);
29        if(weight > 0) {
30          insertEdge(newGraph, j, i, weight);
31        }
32      }
33    }
34    return newGraph;
35  }
36
37  /* checks if the graph is reflexiv */
38  bool isReflexive(Graph *g) {
39    int edgeCount;
40    int i;
41
42    assert (g != NULL);
43    edgeCount = getVerticesCount(g);
44    for (i = 1; i <= edgeCount; i++) {
45      /* every vertex must have a edge to itself */
46      if (getEdgeWeight(g, i, i) == 0)
47        return false;
48    }
49    return true;
50
51  }
52
53  /* checks if the graph is symmetrix
54     every edge must exist in both directions */
55  bool isSymmetric(Graph *g) {
56    int edgeCount;
57    int i, j;
58
59    assert (g != NULL);
60    edgeCount = getVerticesCount(g);
61    for (i = 1; i <= edgeCount; i++) {
62      for (j = 1; j <= edgeCount; j++) {
63        /* if one edge does not have a edge in the other direction the graph is not symmetric*/
64        if (getEdgeWeight(g, i, j) > 0 && getEdgeWeight(g, j, i) == 0) {
65          return false;
66        }
67      }
68    }
69    return true;
70
71  }
72
73  /* checks if the graph is asymmetric */
```

```
74 bool isAsymmetric(Graph *g) {
75  int edgeCount;
76  int i, j;
77 
78   assert (g != NULL);
79   edgeCount = getVerticesCount(g);
80   for (i = 1; i <= edgeCount; i++) {
81     for (j = 1; j <= edgeCount; j++) {
82       /* if one edge exist in both directions the graph is not asymmetric */
83       if (getEdgeWeight(g, i, j) > 0 && getEdgeWeight(g, j, i) > 0) {
84         return false;
85       }
86     }
87   }
88   return true;
89 
90 }
91 
92 /* recursive helper function for isTransitive */
93 bool checkTransitive(Graph *g, int source, int currentNode, bool *visited){
94     int i;
95     int usedIndex;
96 
97     /* on first call use the source node */
98     if (currentNode == 0 )
99       usedIndex = source - 1;
100    else
101      usedIndex = currentNode - 1;
102 
103    /* mark the vertex as visited */
104    visited[usedIndex] = true;
105 
106    /* if there is no edge between the source and the current node the graph is not transitive */
107    if (currentNode != 0 && getEdgeWeight(g, source, currentNode) == 0){
108      return false;
109    }
110 
111    /* visit all vetices from the current vertex and check the edge */
112    for (i = 1; i <= getVerticesCount(g); i++) {
113      if(getEdgeWeight(g, usedIndex + 1, i) > 0 ) {
114        if (!visited[i - 1])
115          return checkTransitive(g, source, i, visited);
116      }
117    }
118    return true;
119 }
120 
121 /* checks if the graph is transitive */
122 bool isTransitive(Graph *g) {
```

```
123   int verticesCount;
124   int i;
125   bool *visited;
126 
127   assert (g != NULL);
128   verticesCount = getVerticesCount(g);
129   /* allocate a array, used for persisting the already visited vertices*/
130   visited = (bool *)malloc(sizeof(bool) * verticesCount);
131   if (visited == NULL) {
132     return false;
133   }
134   /* clear the memory */
135   memset(visited, 0, sizeof(bool) * verticesCount);
136 
137   for (i = 1; i <= verticesCount; i++) {
138     /* check transitiv for all vertices */
139     if (!checkTransitive(g, i, 0, visited)) {
140       /* free the memory */
141       free(visited);
142       return false;
143     }
144     /* clear the memory again */
145     memset(visited, 0, sizeof(bool) * verticesCount);
146   }
147   /* free the memory */
148   free(visited);
149   return true;
150 
151 }
152 
153 /* prints the graph properties */
154 void printGraphProperties(Graph *g) {
155   printf("Graph properties:\n");
156   printf("-----------------\n");
157   printf("Reflexive %d\n", isReflexive(g));
158   printf("Symmetric %d\n", isSymmetric(g));
159   printf("Asymmetric %d\n", isAsymmetric(g));
160   printf("Transitive %d\n", isTransitive(g));
161 }
```

```
1 /*****************************************************************************
2   graph.c
3   Roman Lumetsberger
4 
5   testprogramm from graph functions
6 *****************************************************************************/
7 #include <stdlib.h>
8 #include <stdio.h>
```

```c
#include "dg_adt.h"
#include "graph_algs.h"

int main(int argc, char *argv[]) {
  int testcase;
  Graph *g, *inv;

  if (argc != 2) {
    printf("wrong paramter\n");
    printf("Usage %s testcase\n", argv[0]);
    return EXIT_FAILURE;
  }

  testcase = atoi(argv[1]);
  g = createGraph(4);

  switch (testcase) {
    /* invert */
    case 0:
      insertEdge(g, 1, 2, 2);
      insertEdge(g, 3, 4, 2);
      insertEdge(g, 1, 4, 2);
      break;
    /* reflexiv */
    case 1:
      insertEdge(g, 1, 1, 2);
      insertEdge(g, 2, 2, 2);
      insertEdge(g, 3, 3, 2);
      insertEdge(g, 4, 4, 2);
      break;
    /* symmetric */
    case 2:
      insertEdge(g, 1, 2, 2);
      insertEdge(g, 2, 1, 2);
      insertEdge(g, 3, 4, 2);
      insertEdge(g, 4, 3, 2);
      insertEdge(g, 1, 4, 2);
      insertEdge(g, 4, 1, 2);
      break;

    /* asymmetric */
    case 3:
      insertEdge(g, 1, 2, 2);
      insertEdge(g, 3, 4, 2);
      insertEdge(g, 1, 4, 2);
      break;

    /* transitive */
    case 4:
```

```
58        insertEdge(g, 1, 2, 2);
59        insertEdge(g, 2, 3, 2);
60        insertEdge(g, 1, 3, 2);
61
62        insertEdge(g, 3, 4, 2);
63        insertEdge(g, 1, 4, 2);
64        insertEdge(g, 2, 4, 2);
65        break;
66
67      /*reflexiv, symmetric, transitive */
68      case 5:
69        insertEdge(g, 1, 1, 2);
70        insertEdge(g, 2, 2, 2);
71        insertEdge(g, 3, 3, 2);
72        insertEdge(g, 4, 4, 2);
73
74        insertEdge(g, 1, 2, 2);
75        insertEdge(g, 2, 1, 2);
76
77        insertEdge(g, 2, 3, 2);
78        insertEdge(g, 3, 2, 2);
79
80        insertEdge(g, 1, 3, 2);
81        insertEdge(g, 3, 1, 2);
82
83        insertEdge(g, 3, 4, 2);
84        insertEdge(g, 4, 3, 2);
85        insertEdge(g, 1, 4, 2);
86        insertEdge(g, 4, 1, 2);
87        insertEdge(g, 2, 4, 2);
88        insertEdge(g, 4, 2, 2);
89        break;
90      default:
91        printf("Invalid testcase\n");
92        return EXIT_FAILURE;
93        break;
94    }
95
96    printGraph(g);
97    printf("\n");
98    if (testcase == 0) {
99      inv = invert(g);
100     printGraph(inv);
101     /* set to null, because pointer is not valid anymore */
102     freeGraph(inv);
103     inv = NULL;
104   }
105   else {
106     printGraphProperties(g);
```

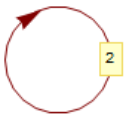
```
107   }
108   freeGraph(g);
109   /* set to null, because pointer is not valid anymore */
110   g = NULL;
111   return EXIT_SUCCESS;
112 }
```

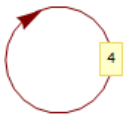
2.3 Testfälle

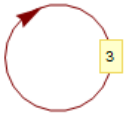
2.3.1 Testfall 1 - invers

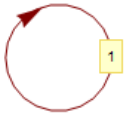
```
romanlum@ubuntu: ~/swo3/UebungMoodle3/Beispiel2
romanlum@ubuntu:~/swo3/UebungMoodle3/Beispiel2$ ./graphm 0
        1    2    3    4
-------------------------
 1 |    0    2    0    2
 2 |    0    0    0    0
 3 |    0    0    0    2
 4 |    0    0    0    0

        1    2    3    4
-------------------------
 1 |    0    0    0    0
 2 |    2    0    0    0
 3 |    0    0    0    0
 4 |    2    0    2    0
romanlum@ubuntu:~/swo3/UebungMoodle3/Beispiel2$ ./graphl 0
1 --> 4(   2) --> 2(   2)
2
3 --> 4(   2)
4

1
2 --> 1(   2)
3
4 --> 3(   2) --> 1(   2)
romanlum@ubuntu:~/swo3/UebungMoodle3/Beispiel2$
```

2.3.2 Testfall 2 - reflexiv










```
romanlum@ubuntu: ~/swo3/UebungMoodle3/Beispiel2

romanlum@ubuntu:~/swo3/UebungMoodle3/Beispiel2$ ./graphm 1
        1    2    3    4
-------------------------
 1 |    2    0    0    0
 2 |    0    2    0    0
 3 |    0    0    2    0
 4 |    0    0    0    2

Graph properties:
-----------------
Reflexive 1
Symmetric 1
Asymmetric 0
Transitive 1
romanlum@ubuntu:~/swo3/UebungMoodle3/Beispiel2$ ./graphl 1
1 --> 1(   2)
2 --> 2(   2)
3 --> 3(   2)
4 --> 4(   2)

Graph properties:
-----------------
Reflexive 1
Symmetric 1
Asymmetric 0
Transitive 1
romanlum@ubuntu:~/swo3/UebungMoodle3/Beispiel2$
```

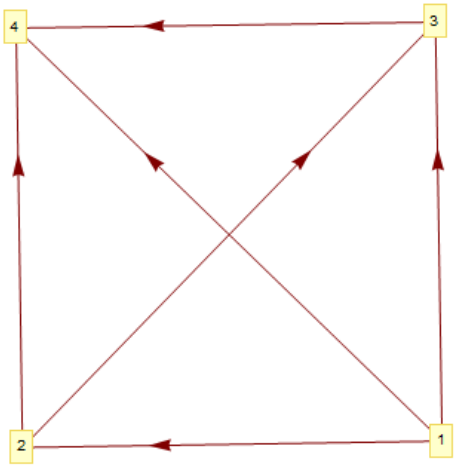
2.3.3 Testfall 3 - symmetrisch


```
romanlum@ubuntu: ~/swo3/UebungMoodle3/Beispiel2
romanlum@ubuntu:~/swo3/UebungMoodle3/Beispiel2$ ./graphm 2
        1    2    3    4
-------------------------
 1 |    0    2    0    2
 2 |    2    0    0    0
 3 |    0    0    0    2
 4 |    2    0    2    0

Graph properties:
-----------------
Reflexive 0
Symmetric 1
Asymmetric 0
Transitive 0
romanlum@ubuntu:~/swo3/UebungMoodle3/Beispiel2$ ./graphl 2
1 --> 4(   2) --> 2(   2)
2 --> 1(   2)
3 --> 4(   2)
4 --> 1(   2) --> 3(   2)

Graph properties:
-----------------
Reflexive 0
Symmetric 1
Asymmetric 0
Transitive 0
romanlum@ubuntu:~/swo3/UebungMoodle3/Beispiel2$
```

2.3.4 Testfall 4 - asymmetrisch

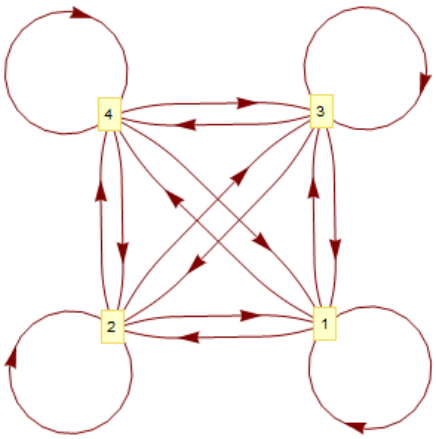

```
romanlum@ubuntu: ~/swo3/UebungMoodle3/Beispiel2
romanlum@ubuntu:~/swo3/UebungMoodle3/Beispiel2$ ./graphm 3
        1    2    3    4
-------------------------
 1 |    0    2    0    2
 2 |    0    0    0    0
 3 |    0    0    0    2
 4 |    0    0    0    0

Graph properties:
-----------------
Reflexive 0
Symmetric 0
Asymmetric 1
Transitive 1
romanlum@ubuntu:~/swo3/UebungMoodle3/Beispiel2$ ./graphl 3
1 --> 4(   2) --> 2(   2)
2
3 --> 4(   2)
4

Graph properties:
-----------------
Reflexive 0
Symmetric 0
Asymmetric 1
Transitive 1
romanlum@ubuntu:~/swo3/UebungMoodle3/Beispiel2$
```

2.3.5 Testfall 5 - transitiv




```
romanlum@ubuntu: ~/swo3/UebungMoodle3/Beispiel2
romanlum@ubuntu:~/swo3/UebungMoodle3/Beispiel2$ ./graphm 4
        1    2    3    4
-------------------------
 1 |    0    2    2    2
 2 |    0    0    2    2
 3 |    0    0    0    2
 4 |    0    0    0    0

Graph properties:
-----------------
Reflexive 0
Symmetric 0
Asymmetric 1
Transitive 1
romanlum@ubuntu:~/swo3/UebungMoodle3/Beispiel2$ ./graphl 4
1 --> 4(   2) --> 3(   2) --> 2(   2)
2 --> 4(   2) --> 3(   2)
3 --> 4(   2)
4

Graph properties:
-----------------
Reflexive 0
Symmetric 0
Asymmetric 1
Transitive 1
romanlum@ubuntu:~/swo3/UebungMoodle3/Beispiel2$
```

2.3.6 Testfall 6 - reflexiv, symmetrisch, transitiv




```
romanlum@ubuntu: ~/swo3/UebungMoodle3/Beispiel2
romanlum@ubuntu:~/swo3/UebungMoodle3/Beispiel2$ ./graphm 5
        1    2    3    4
-------------------------
 1 |    2    2    2    2
 2 |    2    2    2    2
 3 |    2    2    2    2
 4 |    2    2    2    2

Graph properties:
-----------------
Reflexive 1
Symmetric 1
Asymmetric 0
Transitive 1
romanlum@ubuntu:~/swo3/UebungMoodle3/Beispiel2$ ./graphl 5
1 --> 4(   2) --> 3(   2) --> 2(   2) --> 1(   2)
2 --> 4(   2) --> 3(   2) --> 1(   2) --> 2(   2)
3 --> 4(   2) --> 1(   2) --> 2(   2) --> 3(   2)
4 --> 2(   2) --> 1(   2) --> 3(   2) --> 4(   2)

Graph properties:
-----------------
Reflexive 1
Symmetric 1
Asymmetric 0
Transitive 1
romanlum@ubuntu:~/swo3/UebungMoodle3/Beispiel2$
```