Graph

Generated by Doxygen 1.8.11

Sun Dec 31 2017 22:19:06

ii CONTENTS

Contents

1	Todo	o List	2
2	Bug	List	2
3	Data	a Structure Index	2
	3.1	Data Structures	2
4	File	Index	2
	4.1	File List	2
5	Data	a Structure Documentation	3
	5.1	deg_t Struct Reference	3
		5.1.1 Detailed Description	3
		5.1.2 Field Documentation	3
	5.2	graph_t Struct Reference	4
		5.2.1 Detailed Description	4
		5.2.2 Field Documentation	4
	5.3	gvertex_t Struct Reference	5
		5.3.1 Detailed Description	5
		5.3.2 Field Documentation	6
	5.4	info_t Struct Reference	6
		5.4.1 Detailed Description	6
		5.4.2 Field Documentation	6
	5.5	queue_t Struct Reference	7
		5.5.1 Detailed Description	7
		5.5.2 Field Documentation	7
	5.6	stack_t Struct Reference	8
		5.6.1 Detailed Description	8
		5.6.2 Field Documentation	8

6	File I	Documentation	9
	6.1	graphs.c File Reference	9
		6.1.1 Detailed Description	10
		6.1.2 Function Documentation	10
	6.2	graphs.c	17
	6.3	graphs.h File Reference	20
		6.3.1 Detailed Description	21
		6.3.2 Enumeration Type Documentation	22
		6.3.3 Function Documentation	22
	6.4	graphs.h	29
	6.5	queue.c File Reference	29
		6.5.1 Detailed Description	30
		6.5.2 Function Documentation	30
	6.6	queue.c	33
	6.7	queue.h File Reference	34
		6.7.1 Detailed Description	35
		6.7.2 Function Documentation	35
	6.8	queue.h	38
	6.9	stack.c File Reference	38
		6.9.1 Detailed Description	39
		6.9.2 Function Documentation	39
	6.10	stack.c	42
	6.11	stack.h File Reference	42
		6.11.1 Detailed Description	43
		6.11.2 Function Documentation	43
	6.12	stack.h	46
Inc	lex		47

1 Todo List

Class queue_t

 $\textit{base} \, \texttt{should'nt} \, \texttt{be} \, \texttt{accessible}, \\ \texttt{see} \, \texttt{https://stackoverflow.com/questions/5368028/how-to-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-make-struct-$

2 Bug List

Global graph_clone (graph_t *g)

sizeof(pt) is wrong!!

3 Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

deg_	<u>.</u> t	3
grap	h_t	1
gver	tex_t	5
info_	<u>t</u>	5
queu	ue_t	7
stacl	k_t	3

4 File Index

4.1 File List

Here is a list of all files with brief descriptions:

graphs.c Graph's basic operations implementation	9
graphs.h Graph definition and basic operations	20
queue.c Queue's basic operations implementation (using dynamic array)	29
queue.h Queue (using array) definition and basic operations	34
stack.c Stack's basic operations implementation (using dynamic array)	38

stack.h Stack definition and basic operations	42
5 Data Structure Documentation	
5.1 deg_t Struct Reference	
<pre>#include <graphs.h></graphs.h></pre>	
Data Fields	
int inint out	
5.1.1 Detailed Description	
Definition at line 23 of file graphs.h.	
5.1.2 Field Documentation	
5.1.2.1 int in	
indegree	
Definition at line 24 of file graphs.h.	
5.1.2.2 int out	
outdegree	

The documentation for this struct was generated from the following file:

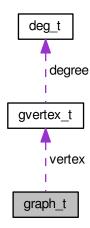
• graphs.h

Definition at line 25 of file graphs.h.

5.2 graph_t Struct Reference

#include <graphs.h>

Collaboration diagram for graph_t:



Data Fields

- gtype_t type
- int n
- int m
- gvertex_t ** vertex

5.2.1 Detailed Description

Definition at line 34 of file graphs.h.

5.2.2 Field Documentation

5.2.2.1 int m

number of edges

Definition at line 37 of file graphs.h.

5.2.2.2 int n

number of vertices

Definition at line 36 of file graphs.h.

5.2.2.3 gtype_t type

graph type: undirected or directed

Definition at line 35 of file graphs.h.

5.2.2.4 gvertex_t** vertex

list of vertices

Definition at line 38 of file graphs.h.

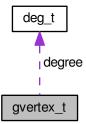
The documentation for this struct was generated from the following file:

• graphs.h

5.3 gvertex_t Struct Reference

```
#include <graphs.h>
```

Collaboration diagram for gvertex_t:



Data Fields

- int * adj list
- int adj_list_len
- deg_t degree

5.3.1 Detailed Description

Definition at line 28 of file graphs.h.

```
5.3.2 Field Documentation
5.3.2.1 int* adj_list
adjacency list (list of vertices index)
Definition at line 29 of file graphs.h.
5.3.2.2 int adj_list_len
sizeof(int) * adj_list_len bytes is reserved for adj_list
Definition at line 30 of file graphs.h.
5.3.2.3 deg_t degree
indegree and outdegree
Definition at line 31 of file graphs.h.
The documentation for this struct was generated from the following file:
    • graphs.h
5.4 info_t Struct Reference
#include <graphs.h>
Data Fields
    • int src
    int * pred

    int * dist

5.4.1 Detailed Description
Definition at line 41 of file graphs.h.
5.4.2 Field Documentation
5.4.2.1 int* dist
Definition at line 44 of file graphs.h.
5.4.2.2 int* pred
Definition at line 43 of file graphs.h.
```

5.4.2.3 int src

Definition at line 42 of file graphs.h.

The documentation for this struct was generated from the following file:

• graphs.h

5.5 queue_t Struct Reference

```
#include <queue.h>
```

Data Fields

- size_t width
- int front
- · int count
- void ** base
- int max_size

5.5.1 Detailed Description

Abstract queue using array.

Todo base should'nt be accessible, see https://stackoverflow.com/questions/5368028/how-to-make-struc

Definition at line 22 of file queue.h.

5.5.2 Field Documentation

5.5.2.1 void** base

pointer to the array

Definition at line 26 of file queue.h.

5.5.2.2 int count

count element amount

Definition at line 25 of file queue.h.

5.5.2.3 int front

front element index

Definition at line 24 of file queue.h.

```
5.5.2.4 int max_size
width \ast max_size bytes is reserved for the queue
Definition at line 27 of file queue.h.
5.5.2.5 size_t width
element size (in bytes)
Definition at line 23 of file queue.h.
The documentation for this struct was generated from the following file:
    • queue.h
5.6 stack_t Struct Reference
#include <stack.h>
Data Fields

    size_t width

    int top

    void ** base
    • int mem_size
5.6.1 Detailed Description
Abstract stack using dynamic array.
Definition at line 20 of file stack.h.
5.6.2 Field Documentation
5.6.2.1 void** base
pointer to the dynamic array
Definition at line 23 of file stack.h.
5.6.2.2 int mem_size
width * mem_size bytes is reserved for the dynamic array
Definition at line 24 of file stack.h.
```

6 File Documentation 9

5.6.2.3 int top

top element index

Definition at line 22 of file stack.h.

5.6.2.4 size_t width

element size (in bytes)

Definition at line 21 of file stack.h.

The documentation for this struct was generated from the following file:

• stack.h

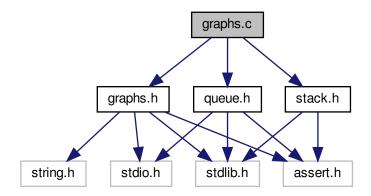
6 File Documentation

6.1 graphs.c File Reference

graph's basic operations implementation

```
#include "graphs.h"
#include "queue.h"
#include "stack.h"
```

Include dependency graph for graphs.c:



Functions

- graph_t * graph_create (int n, gtype_t type)
- void graph_destruct (graph_t *g)
- int is_adj (graph_t *g, int u, int v)
- void graph_add_edge (graph_t *g, int u, int v)
- void graph_adj_list_print (graph_t *g)
- graph_t * graph_clone (graph_t *g)
- void graph_delete_adj_ele (graph_t *g, int u, int v)
- void graph_dot_output (graph_t *g, char *filename)
- void graph_search_dot_output (graph_t *g, info_t *info, char *filename)
- info_t * bfs (graph_t *g, int src)
- void info_destruct (info_t *info)

6.1.1 Detailed Description

graph's basic operations implementation

Author

Firmin MARTIN

Version

0.1

Date

28/12/2017

Definition in file graphs.c.

6.1.2 Function Documentation

```
6.1.2.1 info_t* bfs ( graph_t * g, int src )
```

Launch a bfs on the given graph and source vertex.

Parameters

g	graph
src	source vertex

Returns

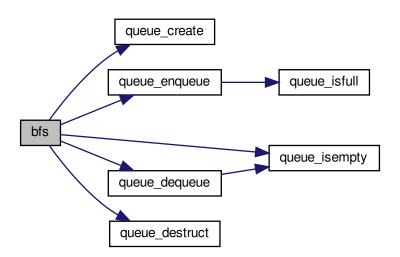
search info which contains for each vertex the distance to the source and the index of its predecessor if it exists (if not, info->pred[i] = -1)

Definition at line 278 of file graphs.c.

```
00278
00279
            assert(src \geq= 0 && src < g\geqn);
            int* color = malloc(g->n * sizeof(int));
int* pred = malloc(g->n * sizeof(int));
int* dist = malloc(g->n * sizeof(int));
00280
00281
00282
            assert(color && pred && dist);

for(int i = 0; i < g->n; i++) {
00283
00284
                color[i] = 0;  /* white
pred[i] = -1;  /* no predecessor
00285
                 pred[i] = -1;
00286
00287
                 dist[i] = g->n + 1; /* no reachable distance */
00288
            color[src] = 1;
00289
00290
            dist[src] = 0;
                                          /* dist(src,src) = 0
00291
            /* int-queue with n slots */
00292
            queue_t* q = queue_create(sizeof(int), g->n);
00293
            queue_enqueue(q, inttoptr(src));
00294
            while(!queue_isempty(q)) {
00295
                int* u = queue_dequeue(q);
for(int i = 0; i < g->vertex[*u]->degree.out; i++) {
00296
00297
                      int v = g->vertex[*u]->adj_list[i];
                      if (color[v] == 0) {
    color[v] = 1; /* gray */
00298
00299
                           dist[v] = dist[*u] + 1;
pred[v] = *u;
00300
00301
00302
                           queue_enqueue(q, inttoptr(v));
00303
00304
00305
                 color[*u] = 2; /* black */
00306
                free(u);
00307
            }
00308
            queue_destruct(q);
00309
            free (color);
00310
            info_t* info = malloc(sizeof(info_t));
00311
            assert(info);
            info->src = src;
info->dist = dist;
00312
00313
            info->pred = pred;
00314
00315
            return info;
00316 }
```

Here is the call graph for this function:



6.1.2.2 void graph_add_edge (graph_t * g, int u, int v)

Add the edge (u, v) in the graph g. If g is an undigraph, (v, u) is also added.

Parameters

g	graph
и	vertex index
V	vertex index

Definition at line 78 of file graphs.c.

```
00078
00079
           gvertex_t *v_u = g->vertex[u], *v_v = g->vertex[v];
assert(u >= 0 && v >= 0 && u < g->n && v < g->n);
08000
00081
           assert(v_u->degree.out <= v_u->adj_list_len);
            if (is_adj(g, u, v)) return;
00083
            if (v_u->degree.out == v_u->adj_list_len) {
00084
                int* newlist = realloc(v_u->adj_list, sizeof(int) * (v_u->
      adj_list_len + 10));
00085
                assert(newlist);
v_u->adj_list = newlist;
00086
00087
                v_u->adj_list_len += 10;
00088
00089
            v_u->adj_list[v_u->degree.out] = v;
00090
           v_u->degree.out++;
00091
            v_v->degree.in++;
00092
            if (g->type == UNDIGRAPH && u != v) {
                assert(v_v->degree.out <= v_v->adj_list_len);
if (v_v->degree.out == v_v->adj_list_len) {
00093
00094
00095
                    int* newlist = realloc(v_v->adj_list, sizeof(int) * (v_v->adj_list_len + 10));
00096
                     assert(newlist);
00097
                    v_v->adj_list = newlist;
v_v->adj_list_len += 10;
00098
00099
00100
                v_v->adj_list[v_v->degree.out] = u;
00101
                v_v->degree.out++;
00102
                v_u->degree.in++;
00103
           g->m++;
00104
00105 }
```

Here is the call graph for this function:



6.1.2.3 void graph_adj_list_print (graph_t * g)

Print graph's adjacency list representation

Parameters



Definition at line 111 of file graphs.c.

00111 {

```
if (g->type == DIGRAPH) printf("type : digraph\n");
             else if (g->type == UNDIGRAPH) printf("type : undigraph\n");
            printf("n=%d, m=%d\n", g->n, g->m);
00114
            for(int i = 0; i < g->n; i++) {
   printf("[%d] ", i);
   if (g->vertex[i]->degree.out > 0) printf("-> ");
00115
00116
00117
                  for(int j = 0; j < g->vertex[i] ->degree.out ; j++) {
    printf("[%d]", g->vertex[i]->adj_list[j]);
00118
00119
00120
                 printf("\n");
00121
            }
00122
00123 }
```

6.1.2.4 graph_t* graph_clone (graph_t * g)

Clone a graph

Parameters

```
g graph
```

Returns

graph cloned

Bug sizeof(pt) is wrong!!

Definition at line 131 of file graphs.c.

```
00131
00132
          graph_t* new_g;
          new_g = (graph_t*) malloc(sizeof(graph_t));
if (new_g != NULL) {
00133
00134
00135
              new_g->type = g->type;
00136
              new_g->n = g->n;
00137
              new_g->m = g->m;
00138
              new_g->vertex = (gvertex_t**) malloc(sizeof(g->vertex));
              assert(new_g->vertex);
for(int i = 0; i < g->n; i++) {
00139
00140
                  new_g->vertex[i] = (gvertex_t*) malloc(sizeof(
00141
      gvertex_t));
00142
                  assert(g->vertex[i]);
00143
                  new_g->vertex[i]->adj_list = (int*) malloc(sizeof(g->
      vertex[i]->adj_list));
00144
                  assert(new_g->vertex[i]->adj_list);
                  new_g->vertex[i]->adj_list_len = g->vertex[i]->
00145
      adj_list_len;
00146
                  new_g->vertex[i]->degree.in = g->vertex[i]->
      degree.in;
00147
                   new_g->vertex[i]->degree.out = g->vertex[i]->
     degree.out;
00148
00149
00150
          return new_g;
00151 }
```

6.1.2.5 graph_t* graph_create (int n, gtype_t type)

Create a graph initialized as a forest with n vertices

Parameters

n	number of vertices	1
type	type of graph (digraph, undigraph)	1

Returns

return a graph initialized as a forest

Definition at line 19 of file graphs.c.

```
00019
00020
           graph_t* g;
           g = (graph_t*) malloc(sizeof(graph_t));
00021
00022
          assert (q);
00023
           if (g != NULL) {
00024
               g->type = type; /* initialize type of graph */
               00025
00026
00027
               assert(g->vertex);
               for(int i = 0; i < g->n; i++) {
    g->vertex[i] = (gvertex_t*) malloc(sizeof(gvertex_t));
00029
00030
                   assert(g->vertex[i]);
00031
                  /* initialize an array with size 5 by default */
g->vertex[i]->adj_list = (int*) malloc(sizeof(int) * 5);
assert(g->vertex[i]->adj_list);
00032
00033
00034
00035
                   g->vertex[i]->adj_list_len = 5;
00036
                   /* g is a forest => deg_in(i) = deg_out(i) = 0 */
00037
                   g->vertex[i]->degree.in = 0;
00038
                   g->vertex[i]->degree.out = 0;
00039
               }
00040
00041
           return g;
00042 }
```

6.1.2.6 void graph_delete_adj_ele (graph_t * g, int u, int v)

Delete the edge (u,v) in graph g. If g is an undigraph, (v,u) is also removed.

Parameters

g	graph
и	vertex index
V	vertex index

Definition at line 159 of file graphs.c.

```
00159
            gvertex_t *v_u = g->vertex[u], *v_v = g->vertex[v];
00160
            assert(v_u && v_v);
int flag = 0;
for (int i = 0; i < v_u->degree.out; i++) {
00161
00162
00163
00164
                 if(v_u->adj_list[i] == v) {
                     flag = 1;
for (int j = i; j < v_u->degree.out - 1; j++) {
    v_u->adj_list[j] = v_u->adj_list[j + 1];
00165
00166
00167
00168
00169
                      v_u->degree.out--;
00170
                }
00171
00172
            if (g->type == UNDIGRAPH && flag) {
                 for (int i = 0; i < v_v->degree.out; i++) {
    if(v_v->adj_list[i] == u) {
00173
00174
00175
                           for (int j = i; j < v_v->degree.out - 1; j++) {
00176
                                v_v-adj_list[j] = v_v-adj_list[j + 1];
00177
00178
                           v_v->degree.out--;
00179
                      }
00180
                 }
00181
00182
            if(flag) g->m--;
00183 }
```

6.1.2.7 void graph_destruct (graph_t * g)

Free a graph

Parameters

```
g a graph
```

Definition at line 48 of file graphs.c.

6.1.2.8 void graph_dot_output (graph_t * g, char * filename)

Output the graph g in dot format (filename.dot).

Parameters

g	graph
filename	filename without any extension

Definition at line 191 of file graphs.c.

```
00191
00192
             FILE* pfile;
              pfile = fopen(filename, "w");
00193
00194
              if (pfile == NULL) perror ("Error opening file");
00195
              if (g->type == DIGRAPH) {
00196
                   fprintf(pfile, "digraph g {\nnode [shape=\"circle\"];\ngraph [overlap=false, concentrate=true];\n")
00197
                   for (int i = 0; i < g->n; i++) {
                        if(g-vertex[i]->degree.out == 0) fprintf(pfile, "%d;\n", i);
for(int j = 0; j < g->vertex[i]->degree.out; j++) {
    fprintf(pfile, "%d -> %d;\n", i, g->vertex[i]->adj_list[j]);
00198
00199
00200
00201
00202
             } else if (g->type == UNDIGRAPH) {
   fprintf(pfile, "graph g {\nnode [shape=\"circle\"];\ngraph [overlap=false, concentrate=true];\n");
   for(int i = 0; i < g->n; i++) {
      if(g->vertex[i]->degree.out == 0) fprintf(pfile, "%d;\n", i);
00203
00204
00205
00206
                         for(int j = 0; j < g->vertex[i]->degree.out ; j++) {
00207
                              if(i <= g->vertex[i]->adj_list[j]) {
    fprintf(pfile, "%d -- %d;\n", i, g->vertex[i]->
00208
00209
adj_list[j]);
00210
                               }
00211
00212
                   }
00213
00214
              fprintf(pfile, "\n");
00215
              fclose(pfile);
00216 }
```

6.1.2.9 void graph_search_dot_output (graph_t * g, info_t * info, char * filename)

Output the graph g with bfs/dfs edge and reachable colored in dot format (filename.dot).

Parameters

g	graph
info	search info, result of a bfs/dfs
filename	filename without any extension

Definition at line 225 of file graphs.c.

```
00225
                                                                                            {
00226
           FILE* pfile;
           pfile = fopen(filename, "w");
00228
            if (pfile == NULL) perror ("Error opening file");
00229
               (g->type == DIGRAPH) {
                 fprintf(pfile, "digraph g {\nnode [shape=\"circle\"];\ngraph [overlap=false, concentrate=true];\n")
00230
00231 ;
                fprintf(pfile, "%d [style=filled fillcolor=red, peripheries=2];\n", info->
                for(int i = 0; i < g->n; i++) {
00232
00233
                     if(g->vertex[i]->degree.out == 0) fprintf(pfile, "%d; \n", i);
00234
                     for(int j = 0; j < g->vertex[i]->degree.out ; j++) {
                          if(info->pred[g->vertex[i]->adj_list[j]] == i) {
    fprintf(pfile, "%d -> %d[color=blue];\n", i, g->vertex[i]->
00235
00236
       adi list[i]);
                               fprintf(pfile, "%d [style=filled fillcolor=red]; \n", g->
       vertex[i]->adj_list[j]);
00238
                          } else fprintf(pfile, "%d -> %d;\n", i, g->vertex[i]->
      adj_list[j]);
00239
00240
                }
00241
            } else if (g->type == UNDIGRAPH) {
00242
                fprintf(pfile, "graph g {\nnode [shape=\"circle\"];\ngraph [overlap=false,concentrate=true];\n");
00243
                fprintf(pfile, "%d [style=filled fillcolor=red, peripheries=2];\n", info->
      src);
00244
                 for (int i = 0; i < g->n; i++) {
                     fif(info->pred[i] == -1) continue;
fprintf(pfile, "%d -- %d[color=blue];\n", info->pred[i], i);
fprintf(pfile, "%d [style=filled fillcolor=red];\n", i);
00245
00247
00248
00249
                 for (int i = 0; i < g->n; i++) {
                     if(g->vertex[i]->degree.out == 0) fprintf(pfile, "%d;\n", i);
for(int j = 0; j < g->vertex[i]->degree.out; j++) {
00250
00251
                          if(i <= g->vertex[i]->adj_list[j]) {
00252
                               if (info->pred(i] == g->vertex(i]->adj_list[j]) continue;
else fprintf(pfile, "%d -- %d;\n", i, g->vertex[i]->
00253
00254
       adj_list[j]);
00255
00256
00257
                }
00258
00259
            fprintf(pfile, "}n");
00260
            fclose(pfile);
00261 }
```

6.1.2.10 void info_destruct (info_t * info)

Free search info

Parameters

```
info search info
```

Definition at line 324 of file graphs.c.

6.2 graphs.c 17

6.1.2.11 int is_adj (graph_t * g, int u, int v)

Determinate if v is in the adjacency list of u in graph g

Parameters

g	graph
и	vertex index
V	vertex index

Definition at line 63 of file graphs.c.

6.2 graphs.c

```
00001
00009 #include "graphs.h"
00010 #include "queue.h"
00011 #include "stack.h"
00012
00019 graph_t* graph_create(int n, gtype_t type) {
00020
           graph_t* g;
00021
           g = (graph_t*) malloc(sizeof(graph_t));
00022
           assert(g);
           if (g != NULL) {
00023
00024
               g->type = type; /* initialize type of graph */
               g->n = n;

g->m = 0;
                              /* initialize number of vertices */
/* g is a forest => 0 edge */
00025
00026
00027
                g->vertex = (gvertex_t**) malloc(sizeof(gvertex_t*) * n);
                assert(g->vertex);
00028
                for(int i = 0; i < g->n; i++) {
    g->vertex[i] = (gvertex_t*) malloc(sizeof(gvertex_t));
00029
00030
00031
                    assert(g->vertex[i]);
00032
                    /* initialize an array with size 5 by default */
00033
                    g->vertex[i]->adj_list = (int*) malloc(sizeof(int) * 5);
00034
                    assert(g->vertex[i]->adj_list);
                    g->vertex[i]->adj_list_len = 5;
/* g is a forest => deg_in(i) = deg_out(i) = 0 */
00035
00036
00037
                    g->vertex[i]->degree.in = 0;
00038
                    g->vertex[i]->degree.out = 0;
00039
               }
00040
00041
           return g;
00042 }
00043
00048 void graph_destruct(graph_t* g) {
00049
        for (int i = 0; i < g -> n; i++) {
00050
                free(g->vertex[i]->adj_list);
00051
                free(g->vertex[i]);
00052
00053
           free (q->vertex);
00054
           free(q);
00055 }
00056
00063 int is_adj(graph_t* g, int u, int v) {
           gvertex_t* v_u = g->vertex[u];
assert(v_u);
for(int i = 0; i < v_u->degree.out; i++) {
00064
00065
00066
00067
               if(v_u->adj_list[i] == v) return 1;
00068
00069
           return 0;
00070 }
00071
00078 void graph_add_edge(graph_t* g, int u, int v) {
           gvertex_t *v_u = g->vertex[u], *v_v = g->vertex[v];
```

```
assert(u \ge 0 \&\& v \ge 0 \&\& u < g \ge n \&\& v < g \ge n);
           assert(v_u->degree.out <= v_u->adj_list_len);
00081
00082
           if (is_adj(g, u, v)) return;
00083
           if (v_u->degree.out == v_u->adj_list_len) {
00084
                int* newlist = realloc(v_u->adj_list, sizeof(int) * (v_u->
      adj_list_len + 10));
00085
               assert (newlist);
00086
                v_u->adj_list = newlist;
00087
                v_u->adj_list_len += 10;
00088
00089
           v_u->adj_list[v_u->degree.out] = v;
00090
           v u->degree.out++;
00091
           v v->degree.in++;
00092
           if (g->type == UNDIGRAPH && u != v) {
00093
                assert(v_v->degree.out <= v_v->adj_list_len);
                if (v_v->degree.out == v_v->adj_list_len) {
00094
                     \verb"int*" newlist = \verb"realloc(v_v->adj_list, sizeof(int) * (v_v->adj_list_len + 10));
00095
00096
                    assert(newlist);
00097
                     v_v->adj_list = newlist;
00098
                    v_v->adj_list_len += 10;
00099
00100
                v_v->adj_list[v_v->degree.out] = u;
00101
                v_v->degree.out++;
00102
                v u->degree.in++;
00103
00104
           g->m++;
00105 }
00106
00111 void graph_adj_list_print(graph_t* g) {
00112    if (g->type == DIGRAPH) printf("type : digraph\n");
00113    else if (g->type == UNDIGRAPH) printf("type : undigraph\n");
           printf("n=%d, m=%d\n", g->n, g->m);
00114
           for(int i = 0; i < g->n; i++) {
    printf("[%d] ", i);
00115
00116
                if (g->vertex[i]->degree.out > 0) printf("-> ");
for(int j = 0; j < g->vertex[i]->degree.out; j++) {
    printf("[%d]", g->vertex[i]->adj_list[j]);
00117
00118
00119
00121
                printf("\n");
00122
           }
00123 }
00124
00131 graph_t* graph_clone(graph_t* g) {
           graph_t* new_g;
new_g = (graph_t*) malloc(sizeof(graph_t));
00132
00133
            if (new_g != NULL) {
00134
00135
                new_g->type = g->type;
00136
                new_g->n = g->n;
                new_g->m = g->m;
00137
                new_q->vertex = (qvertex_t**) malloc(sizeof(q->vertex));
00138
00139
                assert (new_g->vertex);
                for(int i = 0; i < g->n; i++) {
    new_g->vertex[i] = (gvertex_t*) malloc(sizeof()
00140
00141
      gvertex_t));
00142
                    assert(g->vertex[i]);
00143
                    new_g->vertex[i]->adj_list = (int*) malloc(sizeof(g->
       vertex[i]->adj_list));
00144
                    assert(new_g->vertex[i]->adj_list);
                    new_g->vertex[i]->adj_list_len = g->vertex[i]->
00145
      adj_list_len;
00146
                    new q->vertex[i]->degree.in = q->vertex[i]->
      degree.in;
00147
                    new_q->vertex[i]->degree.out = q->vertex[i]->
      degree.out;
00148
00149
00150
           return new_g;
00151 }
00152
00159 void graph_delete_adj_ele(graph_t* g, int u, int v) {
00160
           gvertex_t *v_u = g->vertex[u], *v_v = g->vertex[v];
00161
            assert(v_u && v_v);
00162
           int flag = 0;
           for (int i = 0; i < v_u->degree.out; i++) {
00163
                if(v_u->adj_list[i] == v) {
    flag = 1;
00164
00165
00166
                     for (int j = i; j < v_u->degree.out - 1; j++) {
00167
                         v_u->adj_list[j] = v_u->adj_list[j + 1];
00168
00169
                     v u->degree.out--;
00170
                }
00171
00172
            if (g->type == UNDIGRAPH && flag) {
00173
                 for (int i = 0; i < v_v->degree.out; i++) {
00174
                    if(v_v->adj_list[i] == u) {
                         for (int j = i; j < v_v->degree.out - 1; j++) {
    v_v->adj_list[j] = v_v->adj_list[j + 1];
00175
00176
```

6.2 graphs.c 19

```
00178
                          v_v->degree.out--;
00179
                     }
00180
                }
00181
            if(flag) g->m--;
00182
00183 }
00184
00191 void graph_dot_output(graph_t* g, char* filename) {
00192
           FILE* pfile;
            pfile = fopen(filename, "w");
00193
00194
            if (pfile == NULL) perror ("Error opening file");
            if (g->type == DIGRAPH) {
00195
                 fprintf(pfile, "digraph g {\nnode [shape=\"circle\"];\ngraph [overlap=false, concentrate=true];\n")
00196
00197
                 for (int i = 0; i < g->n; i++) {
                     fif(g-vertex[i]->degree.out == 0) fprintf(pfile, "%d;\n", i);
for(int j = 0; j < g->vertex[i]->degree.out; j++) {
    fprintf(pfile, "%d -> %d;\n", i, g->vertex[i]->adj_list[j]);
00198
00199
00200
00201
00202
00203
            } else if (g->type == UNDIGRAPH) {
                fprintf(pfile, "graph g {\nnode [shape=\"circle\"];\ngraph [overlap=false, concentrate=true];\n");
for(int i = 0; i < g->n; i++) {
00204
00205
00206
                     if(g->vertex[i]->degree.out == 0) fprintf(pfile, "%d;\n", i);
                      for(int j = 0; j < g->vertex[i]->degree.out ; j++) {
00207
00208
                          if(i
                                <= g->vertex[i]->adj_list[j]) {
                               fprintf(pfile, "%d -- %d;\n", i, g->vertex[i]->
00209
      adj_list[j]);
00210
                          }
00211
00212
                }
00213
00214
            fprintf(pfile, "}\n");
00215
            fclose(pfile);
00216 }
00217
00225 void graph_search_dot_output(graph_t* g, info_t* info, char* filename)
00226
            FILE* pfile;
            pfile = fopen(filename, "w");
00227
            if (pfile == NULL) perror ("Error opening file");
00228
00229
            if (a->t,vpe == DTGRAPH) {
00230
                fprintf(pfile, "digraph g {\nnode [shape=\"circle\"];\ngraph [overlap=false, concentrate=true];\n")
00231
                fprintf(pfile, "%d [style=filled fillcolor=red, peripheries=2]; \n", info->
00232
                for (int i = 0; i < g->n; i++) {
                     if(g->vertex[i]->degree.out == 0) fprintf(pfile, "%d;\n", i);
00233
                     for(int j = 0; j < g->vertex[i]->degree.out; j++) {
    if(info->pred[g->vertex[i]->adj_list[j]] == i) {
00234
00235
                               fprintf(pfile, "%d -> %d[color=blue];\n", i, g->vertex[i]->
00236
       adj_list[j]);
00237
                               fprintf(pfile, "%d [style=filled fillcolor=red]; \n", g->
      vertex[i]->adj_list[j]);
00238
                          } else fprintf(pfile, "%d -> %d;\n", i, g->vertex[i]->
      adj_list[j]);
00239
00240
            } else if (g->type == UNDIGRAPH) {
   fprintf(pfile, "graph g {\nnode [shape=\"circle\"];\ngraph [overlap=false,concentrate=true];\n");
   fprintf(pfile, "%d [style=filled fillcolor=red, peripheries=2];\n", info->
00241
00242
00243
      src);
00244
                 for (int i = 0; i < g -> n; i++) {
                     if(info->pred[i] == -1) continue;
fprintf(pfile, "%d -- %d[color=blue];\n", info->pred[i], i);
fprintf(pfile, "%d [style=filled fillcolor=red];\n", i);
00245
00246
00247
00248
00249
                 for (int i = 0; i < q > n; i++) {
                     if(g->vertex[i]->degree.out == 0) fprintf(pfile, "%d;\n", i);
00250
00251
                      for(int j = 0; j < g->vertex[i]->degree.out ; j++) {
00252
                          if(i <= g->vertex[i]->adj_list[j]) {
                               if (info->pred[i] == g->vertex[i]->adj_list[j]) continue;
else fprintf(pfile, "%d -- %d;\n", i, g->vertex[i]->
00253
00254
       adj_list[j]);
00255
00256
                     }
00257
00258
            fprintf(pfile, "}\n");
00259
00260
            fclose(pfile);
00261 }
00262
00263 static int* inttoptr(int i) {
00264
           int* ptr = malloc(sizeof(int));
00265
            assert(ptr);
00266
           *ptr = i:
```

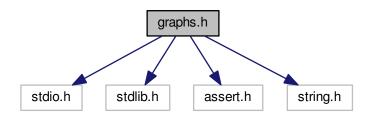
```
00267
            return ptr;
00268 }
00269
00278 info_t* bfs(graph_t* g, int src) {
00279     assert(src >= 0 && src < g->n);
00280     int* color = malloc(g->n * sizeof(int));
            int* pred = malloc(g > n * sizeof(int));
int* dist = malloc(g - > n * sizeof(int));
00282
00283
            assert(color && pred && dist);
            00284
00285
00286
                 dist[i] = g->n + 1; /* no reachable distance */
00287
00288
00289
            color[src] = 1;
00290
            dist[src] = 0;
                                          /* dist(src,src) = 0
            /* int-queue with n slots */
queue_t* q = queue_create(sizeof(int), g->n);
queue_enqueue(q, inttoptr(src));
00291
00292
00293
00294
            while(!queue_isempty(q)) {
00295
                 int* u = queue_dequeue(q);
                 for(int i = 0; i < g->vertex[*u]->degree.out; i++) {
  int v = g->vertex[*u]->adj_list[i];
00296
00297
00298
                      if (color[v] == 0) {
    color[v] = 1; /* gray */
00299
00300
                           dist[v] = dist[*u] + 1;
pred[v] = *u;
00301
00302
                           queue_enqueue(q, inttoptr(v));
00303
00304
                 }
00305
                 color[*u] = 2; /* black */
00306
                 free(u);
00307
00308
            queue_destruct(q);
00309
            free(color);
            info_t* info = malloc(sizeof(info_t));
00310
00311
            assert(info);
00312
            info->src = src;
00313
            info->dist = dist;
00314
            info->pred = pred;
00315
            return info;
00316 }
00317
00318
00324 void info_destruct(info_t* info) {
00325
            free(info->dist);
00326
            free(info->pred);
00327
            free(info);
00328 }
```

6.3 graphs.h File Reference

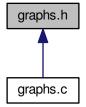
graph definition and basic operations

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
```

Include dependency graph for graphs.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct deg t
- struct gvertex_t
- struct graph_t
- struct info_t

Enumerations

enum gtype_t { UNDIGRAPH, DIGRAPH }

Functions

```
graph_t * graph_clone (graph_t *g)
```

- graph_t * graph_create (int n, gtype_t type)
- void graph_add_edge (graph_t *g, int u, int v)
- void graph_adj_list_print (graph_t *g)
- void graph_delete_adj_ele (graph_t *g, int u, int v)
- void graph_destruct (graph_t *g)
- void graph_dot_output (graph_t *g, char *filename)
- void graph_search_dot_output (graph_t *g, info_t *info, char *filename)
- int is_adj (graph_t *g, int u, int v)
- info_t * bfs (graph_t *g, int src)
- void info_destruct (info_t *info)

6.3.1 Detailed Description

graph definition and basic operations

Author

Firmin MARTIN

Version

0.1

Date

28/12/2017

Definition in file graphs.h.

6.3.2 Enumeration Type Documentation

6.3.2.1 enum gtype t

Enumerator

UNDIGRAPH undirect graph
DIGRAPH direct graph

Definition at line 18 of file graphs.h.

```
00018 {
00019 UNDIGRAPH,
00020 DIGRAPH
00021 } gtype_t;
```

6.3.3 Function Documentation

```
6.3.3.1 info_t* bfs ( graph_t * g, int src )
```

Launch a bfs on the given graph and source vertex.

Parameters

g	graph
src	source vertex

Returns

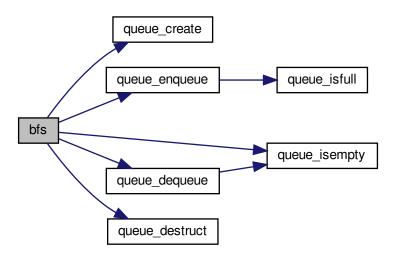
search info which contains for each vertex the distance to the source and the index of its predecessor if it exists (if not, info->pred[i] = -1)

Definition at line 278 of file graphs.c.

```
00279
            assert(src >= 0 && src < g->n);
            int* color = malloc(g->n * sizeof(int));
int* pred = malloc(g->n * sizeof(int));
int* dist = malloc(g->n * sizeof(int));
00280
00281
00282
           00283
00284
00285
00286
                 dist[i] = g->n + 1; /* no reachable distance */
00287
00288
            color[src] = 1;
00289
00290
            dist[src] = 0;
                                          /* dist(src,src) = 0
            /* int-queue with n slots */
00291
00292
            queue_t* q = queue_create(sizeof(int), g->n);
00293
            queue_enqueue(q, inttoptr(src));
00294
            while(!queue_isempty(q)) {
                 int* u = queue_dequeue(q);
for(int i = 0; i < g->vertex[*u]->degree.out; i++) {
   int v = g->vertex[*u]->adj_list[i];
00295
00296
00297
                      if (color[v] == 0) {
    color[v] = 1; /* gray */
00298
00299
                          dist[v] = dist[*u] + 1;
pred[v] = *u;
00300
00301
00302
                           queue_enqueue(q, inttoptr(v));
00303
                      }
00304
                 }
```

```
color[*u] = 2; /* black */
00306
                free(u);
00307
00308
           queue_destruct(q);
00309
           free(color);
info_t* info = malloc(sizeof(info_t));
00310
00311
           assert(info);
00312
           info->src = src;
           info->dist = dist;
info->pred = pred;
00313
00314
           return info;
00315
00316 }
```

Here is the call graph for this function:



6.3.3.2 void graph_add_edge (graph_t * g, int u, int v)

Add the edge (u,v) in the graph g. If g is an undigraph, (v,u) is also added.

Parameters

g	graph
и	vertex index
V	vertex index

Definition at line 78 of file graphs.c.

```
00078
           gvertex_t *v_u = g->vertex[u], *v_v = g->vertex[v];
assert(u >= 0 && v >= 0 && u < g->n && v < g->n);
00079
08000
00081
           assert(v_u->degree.out <= v_u->adj_list_len);
00082
           if (is_adj(g, u, v)) return;
00083
           if (v_u->degree.out == v_u->adj_list_len) {
               int* newlist = realloc(v_u->adj_list, sizeof(int) * (v_u->
00084
      adj_list_len + 10));
00085
              assert(newlist);
00086
                v_u->adj_list = newlist;
```

```
v_u->adj_list_len += 10;
00088
00089
          v_u->adj_list[v_u->degree.out] = v;
00090
          v_u->degree.out++;
00091
          v_v->degree.in++;
          if (q->type == UNDIGRAPH && u != v) {
00092
00093
              assert(v_v->degree.out <= v_v->adj_list_len);
00094
              if (v_v->degree.out == v_v->adj_list_len) {
00095
                  int* newlist = realloc(v_v->adj_list, sizeof(int) * (v_v->adj_list_len + 10));
00096
                  assert(newlist);
                  v_v->adj_list = newlist;
v_v->adj_list_len += 10;
00097
00098
00099
00100
              v_v->adj_list[v_v->degree.out] = u;
00101
              v_v->degree.out++;
00102
              v_u->degree.in++;
00103
00104
          g->m++;
00105 }
```

Here is the call graph for this function:



6.3.3.3 void graph_adj_list_print (graph_t * g)

Print graph's adjacency list representation

Parameters

```
g graph
```

Definition at line 111 of file graphs.c.

```
00111
00112
                 if (g->type == DIGRAPH) printf("type : digraph\n");
                 else if (g->type == UNDIGRAPH) printf("type : undigraph\n");
printf("n=%d, m=%d\n", g->n, g->m);
for(int i = 0; i < g->n; i++) {
    printf("[%d] ", i);
    if (g->vertex[i]->degree.out > 0) printf("-> ");
00113
00114
00115
00116
00117
                        for(int j = 0; j < g->vertex[i]->degree.out ; j++) {
   printf("[%d]", g->vertex[i]->adj_list[j]);
00118
00119
00120
                        printf("\n");
00121
00122
                 }
00123 }
```

6.3.3.4 graph_t* graph_clone (graph_t * g)

Clone a graph

Parameters



Returns

graph cloned

Bug sizeof(pt) is wrong!!

Definition at line 131 of file graphs.c.

```
00131
00132
           graph_t* new_g;
new_g = (graph_t*) malloc(sizeof(graph_t));
00133
00134
           if (new_g != NULL) {
00135
               new_g->type = g->type;
               new_g->n = g->n;
new_g->m = g->m;
new_g->vertex = (gvertex_t**) malloc(sizeof(g->vertex));
00136
00137
00138
               assert(new_g->vertex);
for(int i = 0; i < g->n; i++) {
   new_g->vertex[i] = (gvertex_t*) malloc(sizeof())
00139
00141
      gvertex_t));
00142
           assert(g->vertex[i]);
      new_g->vertex[i]->adj_list = (int*) malloc(sizeof(g->
vertex[i]->adj_list));
00143
00144
                    assert(new_g->vertex[i]->adj_list);
00145
                    new_g->vertex[i]->adj_list_len = g->vertex[i]->
       adj_list_len;
00146
                     new_g->vertex[i]->degree.in = g->vertex[i]->
      degree.in;
00147
                     new_g->vertex[i]->degree.out = g->vertex[i]->
00148
00149
00150
           return new_g;
00151 }
```

6.3.3.5 graph_t* graph_create (int n, gtype_t type)

Create a graph initialized as a forest with n vertices

Parameters

n	number of vertices
type	type of graph (digraph, undigraph)

Returns

return a graph initialized as a forest

Definition at line 19 of file graphs.c.

```
g->m = 0;
                                  /* g is a forest => 0 edge */
               g->vertex = (gvertex_t**) malloc(sizeof(gvertex_t*) * n);
00027
00028
               assert(g->vertex);
               for(int i = 0; i < g->n; i++) {
   g->vertex[i] = (gvertex_t*) malloc(sizeof(gvertex_t));
00029
00030
00031
                   assert(q->vertex[i]);
                   /\star initialize an array with size 5 by default \star/
00033
                   g->vertex[i]->adj_list = (int*) malloc(sizeof(int) * 5);
00034
                    assert(g->vertex[i]->adj_list);
00035
                   g->vertex[i]->adj_list_len = 5;
                   /* g is a forest => deg_in(i) = deg_out(i) = 0 */
g->vertex[i]->degree.in = 0;
00036
00037
00038
                    g->vertex[i]->degree.out = 0;
00039
00040
00041
           return g;
00042 }
```

6.3.3.6 void graph_delete_adj_ele (graph_t * g, int u, int v)

Delete the edge (u,v) in graph g. If g is an undigraph, (v,u) is also removed.

Parameters

g	graph
и	vertex index
V	vertex index

Definition at line 159 of file graphs.c.

```
00159
00160
          gvertex_t *v_u = g->vertex[u], *v_v = g->vertex[v];
00161
          assert(v_u && v_v);
00162
          int flag = 0;
          for (int i = 0; i < v_u->degree.out; i++) {
00163
               if (v_u->adj_list[i] == v) {
00164
00165
                   flag = 1;
                   for (int j = i; j < v_u->degree.out - 1; j++) {
00166
00167
                        v_u->adj_list[j] = v_u->adj_list[j + 1];
00168
00169
                   v_u->degree.out--;
00170
               }
00171
00172
           if (g->type == UNDIGRAPH && flag) {
00173
               for (int i = 0; i < v_v->degree.out; i++) {
00174
                   if(v_v->adj_list[i] == u) {
                        for (int j = i; j < v_v->degree.out - 1; j++) {
    v_v->adj_list[j] = v_v->adj_list[j + 1];
00175
00176
00177
00178
                        v_v->degree.out--;
00179
                   }
00180
               }
00181
           if(flag) g->m--;
00182
00183 }
```

6.3.3.7 void graph_destruct (graph_t * g)

Free a graph

Parameters

g a graph

Definition at line 48 of file graphs.c.

6.3.3.8 void graph_dot_output (graph_t * g, char * filename)

Output the graph g in dot format (filename.dot).

Parameters

g	graph
filename	filename without any extension

Definition at line 191 of file graphs.c.

```
00191
00192
           FILE* pfile;
00193
           pfile = fopen(filename, "w");
00194
           if (pfile == NULL) perror ("Error opening file");
00195
           if (g->type == DIGRAPH) {
00196
                fprintf(pfile, "digraph g {\nnode [shape=\"circle\"];\ngraph [overlap=false, concentrate=true];\n")
00197
                for (int i = 0; i < g->n; i++) {
                    if(g->vertex[i] ->degree.out == 0) fprintf(pfile, "%d;\n", i);
for(int j = 0; j < g->vertex[i] ->degree.out; j++) {
00198
00199
00200
                         fprintf(pfile, "%d -> %d; \n", i, g->vertex[i]->adj_list[j]);
00201
00202
00203
           } else if (g->type == UNDIGRAPH) {
               fprintf(pfile, "graph g {\nnode [shape=\"circle\"];\ngraph [overlap=false, concentrate=true];\n");
for(int i = 0; i < g->n; i++) {
00204
00205
00206
                     if(g->vertex[i]->degree.out == 0) fprintf(pfile, "%d;\n", i);
                     for(int j = 0; j < g->vertex[i]->degree.out ; j++) {
00207
                         if(i <= g->vertex[i]->adj_list[j]) {
   fprintf(pfile, "%d -- %d;\n", i, g->vertex[i]->
00208
00209
      adj_list[j]);
00210
00211
00212
               }
00213
           fprintf(pfile, "}n");
00214
00215
           fclose(pfile);
00216 }
```

6.3.3.9 void graph_search_dot_output (graph_t * g, info_t * info, char * filename)

Output the graph g with bfs/dfs edge and reachable colored in dot format (filename.dot).

Parameters

g	graph
info	search info, result of a bfs/dfs
filename	filename without any extension

Definition at line 225 of file graphs.c.

00225 {

```
00226
             FILE* pfile;
00227
            pfile = fopen(filename, "w");
             if (pfile == NULL) perror ("Error opening file");
if (g->type == DIGRAPH) {
00228
00229
                  fprintf(pfile, "digraph g {\nnode [shape=\"circle\"];\ngraph [overlap=false, concentrate=true];\n")
00230
00231
                  fprintf(pfile, "%d [style=filled fillcolor=red, peripheries=2];\n", info->
00232
                  for (int i = 0; i < g->n; i++) {
                        if(g->vertex[i]->degree.out == 0) fprintf(pfile, "%d;\n", i);
00233
                       for(int j = 0; j < g->vertex[i]->degree.out ; j++) {
   if(info->pred[g->vertex[i]->adj_list[j]] == i) {
00234
00235
                                 fprintf(pfile, "%d -> %d[color=blue]; \n", i, g->vertex[i]->
00236
       adj_list[j]);
00237
                                 fprintf(pfile, "%d [style=filled fillcolor=red]; n", g->
       vertex[i]->adj_list[j]);
                             } else fprintf(pfile, "%d -> %d;\n", i, g->vertex[i]->
00238
       adj_list[j]);
00239
00240
                  }
             } else if (g->type == UNDIGRAPH) {
   fprintf(pfile, "graph g {\nnode [shape=\"circle\"];\ngraph [overlap=false,concentrate=true];\n");
   fprintf(pfile, "%d [style=filled fillcolor=red, peripheries=2];\n", info->
00241
00242
00243
       src);
00244
                  for (int i = 0; i < q -> n; i++) {
00245
                       f(info->pred[i] == -1) continue;
fprintf(pfile, "%d -- %d[color=blue];\n", info->pred[i], i);
fprintf(pfile, "%d [style=filled fillcolor=red];\n", i);
00246
00247
00248
                  for (int i = 0; i < g->n; i++) {
00249
00250
                       if(g->vertex[i]->degree.out == 0) fprintf(pfile, "%d;\n", i);
                       for(int j = 0; j < g->vertex[i]->degree.out ; j++) {
   if(i <= g->vertex[i]->adj_list[j]) {
00251
00252
                                 if (info->pred[i] == g->vertex[i]->adj_list[j]) continue;
else fprintf(pfile, "%d -- %d;\n", i, g->vertex[i]->
00253
00254
       adj_list[j]);
00255
                             }
00256
00257
                  }
00258
00259
             fprintf(pfile, "\n');
00260
             fclose(pfile);
00261 }
```

6.3.3.10 void info_destruct (info t * info)

Free search info

Parameters

```
info search info
```

Definition at line 324 of file graphs.c.

```
00324
00325     free(info->dist);
00326     free(info->pred);
00327     free(info);
```

6.3.3.11 int is_adj (graph_t * g, int u, int v)

Determinate if v is in the adjacency list of u in graph g

Parameters

g	graph
и	vertex index
V	vertex index

6.4 graphs.h 29

Definition at line 63 of file graphs.c.

6.4 graphs.h

```
00001 #ifndef GRAPHS_H
00002 #define GRAPHS_H
00003
00012 #include <stdio.h>
00013 #include <stdlib.h>
00014 #include <assert.h>
00015 #include <string.h>
00016
00018 typedef enum {
00019
            UNDIGRAPH,
00020
            DIGRAPH
00021 } gtype_t;
00022
00023 typedef struct deg_t {
00024 int in;
00025 int out;
00026 } deg_t;
00027
00028 typedef struct gvertex_t {
00032 } gvertex_t;
00033
00034 typedef struct graph_t {
         gtype_t type;
int n;
00035
00036
           int m;
00037
00038
            gvertex t** vertex:
00039 } graph_t;
00040
00041 typedef struct info_t {
         int src;
int* pred;
int* dist;
00042
00043
00044
00045 } info_t;
00046
00047 graph_t* graph_clone(graph_t* g);
00048 graph_t* graph_create(int n, gtype_t type);

00048 graph_t* graph_acte(int n, gtype_t type);

00049 void graph_add_edge(graph_t* g, int u, int v);

00050 void graph_adj_list_print(graph_t* g);

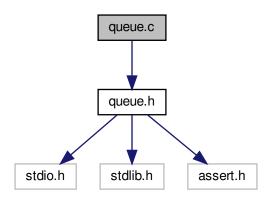
00051 void graph_delete_adj_ele(graph_t* g, int u, int v);
00052 void graph_destruct(graph_t* g);
00053 void graph_dot_output(graph_t* g, char* filename);
00054 void graph_search_dot_output(graph_t* g, info_t* info, char* filename);
00055 int is_adj(graph_t* g, int u, int v);
00056 info_t* bfs(graph_t* g, int src);
00057 void info_destruct(info_t* info);
00058
00059 #endif //GRAPHS_H
```

6.5 queue.c File Reference

queue's basic operations implementation (using dynamic array)

```
#include "queue.h"
```

Include dependency graph for queue.c:



Functions

- int queue_isempty (queue_t *q)
- int queue_isfull (queue_t *q)
- void queue_enqueue (queue_t *q, void *e)
- void * queue_dequeue (queue_t *q)
- queue_t * queue_create (size_t width, int max_size)
- void queue_destruct (queue_t *q)

6.5.1 Detailed Description

queue's basic operations implementation (using dynamic array)

Author

Firmin MARTIN

Version

0.1

Date

28/12/2017

Definition in file queue.c.

6.5.2 Function Documentation

6.5.2.1 queue_t* queue_create (size_t width, int max_size)

Given the size of each element and the queue size, create a queue.

Parameters

width	size of each element
max_size	size of the queue, max_size*width bytes will be reserved (definitively) for the queue

Returns

a queue initialized

Definition at line 71 of file queue.c.

6.5.2.2 void* queue_dequeue (queue_t * q)

Dequeue an element from the queue s.

Parameters

```
q queue
```

Returns

an element or NULL if the queue is empty

Definition at line 54 of file queue.c.

Here is the call graph for this function:



```
6.5.2.3 void queue_destruct ( queue_t * q )
```

Free a queue.

Parameters

```
q a queue
```

Definition at line 88 of file queue.c.

```
00088

00089 free(q->base);

00090 free(q);

00091 }
```

6.5.2.4 void queue_enqueue (queue_t * q, void * e)

Enqueue an element e into the queue q.

Parameters

q	queue
e	element which be enqueued

Definition at line 37 of file queue.c.

Here is the call graph for this function:

```
queue_enqueue queue_isfull
```

6.5.2.5 int queue_isempty (queue_t * q)

Determinate the emptiness of a queue.

6.6 queue.c 33

Parameters

```
s queue
```

Returns

1 if the queue s is empty, 0 otherwise.

Definition at line 17 of file queue.c.

6.5.2.6 int queue_isfull (queue_t * q)

Determinate the fullness of a queue.

Parameters

```
s queue
```

Returns

1 if the queue s is full, 0 otherwise.

Definition at line 27 of file queue.c.

```
00027 {
00028 return q->count == q->max_size;
00029 }
```

6.6 queue.c

```
00001
00009 #include "queue.h"
00010
00017 int queue_isempty(queue_t* q) {
00018
        return q->count == 0;
00019 }
00020
00027 int queue_isfull(queue_t* q) {
00028
       return q->count == q->max_size;
00029 }
00030
00037 void queue_enqueue(queue_t* q, void* e) {
      if(queue_isfull(q)) {
00038
            fprintf(stderr, "The queue is full : failed to enqueue.\n");
00039
00040
00041
        q->base[q->front] = e;
if (q->front == q->max_size - 1) q->front = 0;
00042
00043
00044
        else q->front++;
00045
        q->count = q->count + 1;
00046 }
00047
00054 void* queue_dequeue(queue_t* q) {
00055
      00056
00057
            return NULL;
```

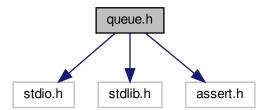
```
00059
           void* e = q->base[(q->front - q->count + q->max_size)%q->
      max_size];
00060
          q->count--;
00061
          return e;
00062 }
00063
00071 queue_t* queue_create(size_t width, int max_size) {
00072
         queue_t* q = malloc(sizeof(queue_t));
00073
          assert(q);
00074
          q->width = width;
00075
          q->max_size = max_size;
q->base = (void**) calloc(q->max_size, sizeof(void*));
00076
00077
          assert (q->base);
          q->front = 0;
q->count = 0;
00078
00079
08000
          return q;
00081 }
00082
00088 void queue_destruct(queue_t* q) {
00089
          free(q->base);
00090
          free(q);
00091 }
00092
00093
00094
```

6.7 queue.h File Reference

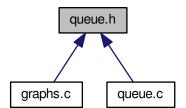
queue (using array) definition and basic operations

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
```

Include dependency graph for queue.h:



This graph shows which files directly or indirectly include this file:



Data Structures

• struct queue_t

Functions

- queue_t * queue_create (size_t width, int max_size)
- void queue_destruct (queue_t *q)
- int queue_isempty (queue_t *q)
- int queue_isfull (queue_t *q)
- void * queue_dequeue (queue_t *q)
- void queue_enqueue (queue_t *q, void *e)

6.7.1 Detailed Description

queue (using array) definition and basic operations

Author

Firmin MARTIN

Version

0.1

Date

28/12/2017

Definition in file queue.h.

6.7.2 Function Documentation

6.7.2.1 queue_t* queue_create (size_t width, int max_size)

Given the size of each element and the queue size, create a queue.

Parameters

width	size of each element
max_size	size of the queue, max_size*width bytes will be reserved (definitively) for the queue

Returns

a queue initialized

Definition at line 71 of file queue.c.

```
00072
           queue_t* q = malloc(sizeof(queue_t));
00073
           assert(q);
00074
           q->width = width;
00075
           q->max_size = max_size;
q->base = (void**) calloc(q->max_size, sizeof(void*));
00076
           assert (q->base);
00077
00078
           q \rightarrow front = 0;
           q->count = 0;
00079
08000
           return q;
00081 }
```

6.7.2.2 void* queue_dequeue (queue_t * q)

Dequeue an element from the queue s.

Parameters



Returns

an element or NULL if the queue is empty

Definition at line 54 of file queue.c.

Here is the call graph for this function:



6.7.2.3 void queue_destruct (queue_t * q)

Free a queue.

Parameters

q a queue

Definition at line 88 of file queue.c.

```
00088

00089 free(q->base);

00090 free(q);

00091 }
```

6.7.2.4 void queue_enqueue (queue_t * q, void * e)

Enqueue an element e into the queue q.

Parameters

Ç	queue
ϵ	element which be enqueued

Definition at line 37 of file queue.c.

Here is the call graph for this function:



6.7.2.5 int queue_isempty ($queue_t * q$)

Determinate the emptiness of a queue.

Parameters



Returns

1 if the queue s is empty, 0 otherwise.

Definition at line 17 of file queue.c.

6.7.2.6 int queue_isfull (queue_t * q)

Determinate the fullness of a queue.

Parameters

```
s queue
```

Returns

1 if the queue s is full, 0 otherwise.

Definition at line 27 of file queue.c.

```
00027 {
00028 return q->count == q->max_size;
00029 }
```

6.8 queue.h

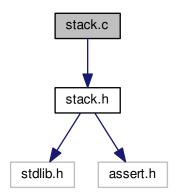
```
00001 #ifndef STACK_H
00002 #define STACK_H
00003
00004 #include <stdio.h>
00005 #include <stdlib.h>
00006 #include <assert.h>
00007
00022 typedef struct {
00023 size_t width;
00024 int front;
            int count;
void** base;
int max_size;
00025
00026
00027
00028 } queue_t;
00030 queue_t* queue_create(size_t width, int max_size);
00031 void queue_destruct(queue_t* q);
00032 int queue_isempty(queue_t* q);
00033 int queue_isfull(queue_t* q);
00034 void* queue_dequeue(queue_t* q);
00035 void queue_enqueue(queue_t* q, void* e);
00036
00037 #endif /* ifndef STACK_H */
```

6.9 stack.c File Reference

stack's basic operations implementation (using dynamic array)

#include "stack.h"

Include dependency graph for stack.c:



Functions

- int stack_isempty (stack_t *s)
- void stack_push (stack_t *s, void *e)
- void * stack_pop (stack_t *s)
- stack_t * stack_create (size_t width)
- void stack_destruct (stack_t *s)

6.9.1 Detailed Description

stack's basic operations implementation (using dynamic array)

Author

Firmin MARTIN

Version

0.1

Date

28/12/2017

Definition in file stack.c.

6.9.2 Function Documentation

6.9.2.1 stack_t* stack_create (size_t width)

Given the size of each element, create a stack 10 * sizeof(void*) bytes is reserved by default.

Parameters

width size of each element

Returns

a stack initialized

Definition at line 57 of file stack.c.

6.9.2.2 void stack_destruct (stack_t * s)

Free a stack.

Parameters

```
s a stack
```

Definition at line 73 of file stack.c.

```
00073

00074 free(s->base);

00075 free(s);
```

6.9.2.3 int stack_isempty ($stack_t * s$)

Determinate the emptiness of a stack.

Parameters

```
s stack
```

Returns

1 if the stack s is empty, 0 otherwise.

Definition at line 17 of file stack.c.

```
6.9.2.4 void* stack_pop ( stack_t * s )
```

Pop out an element from the stack s.

Parameters

```
s stack
```

Returns

an element

Definition at line 44 of file stack.c.

```
00044 {
00045 if (stack_isempty(s)) return NULL;
00046 s->top--;
00047 return s->base[s->top + 1];
```

Here is the call graph for this function:



```
6.9.2.5 void stack_push ( stack_t * s, void * e )
```

Push an element e into the stack s.

Parameters

s	stack
е	element which be pushed

Definition at line 27 of file stack.c.

6.10 stack.c

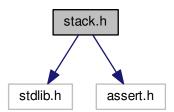
```
00001
00009 #include "stack.h"
00010
00017 int stack_isempty(stack_t* s) {
00018
         return s \rightarrow top == -1;
00019 }
00020
00027 void stack_push(stack_t* s, void* e) {
        s->top++;
00028
          if (s->top == s->mem_size) {
00029
00030
             void** newptr = realloc(s->base, sizeof(void*) * (s->mem_size + 10));
00031
             assert(newptr);
00032
             s->base = newptr;
             s->mem_size += 10;
00033
00034
00035
         s->base[s->top] = e;
00036 }
00037
00044 void* stack_pop(stack_t* s) {
00045
        if (stack_isempty(s)) return NULL;
00046
         s->top--;
00047
         return s->base[s->top + 1];
00048 }
00049
00057 stack_t* stack_create(size_t width) {
       stack_t* s = malloc(sizeof(stack_t));
00058
00059
         assert(s);
         s->width = width;
00060
00061
         s->mem_size = 10;
00062
         s->base = (void**) malloc(sizeof(void*) * s->mem_size);
00063
         assert(s->base);
00064
         s->top = -1;
00065
         return s:
00066 }
00067
00073 void stack_destruct(stack_t* s) {
00074
         free(s->base);
00075
         free(s);
00076 }
00077
00078
00079
```

6.11 stack.h File Reference

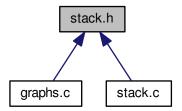
stack definition and basic operations

```
#include <stdlib.h>
#include <assert.h>
```

Include dependency graph for stack.h:



This graph shows which files directly or indirectly include this file:



Data Structures

struct stack_t

Functions

- stack_t * stack_create (size_t width)
- void stack_destruct (stack_t *s)
- int stack_isempty (stack_t *s)
- void * stack_pop (stack_t *s)
- void stack_push (stack_t *s, void *e)

6.11.1 Detailed Description

stack definition and basic operations

Author

Firmin MARTIN

Version

0.1

Date

28/12/2017

Definition in file stack.h.

6.11.2 Function Documentation

6.11.2.1 stack_t* stack_create (size_t width)

Given the size of each element, create a stack 10 * sizeof(void*) bytes is reserved by default.

Parameters

width size of each element

Returns

a stack initialized

Definition at line 57 of file stack.c.

```
00057
          stack_t* s = malloc(sizeof(stack_t));
00058
          assert(s);
00059
00060
          s->width = width;
00061
          s->mem_size = 10;
          s->base = (void**) malloc(sizeof(void*) * s->mem_size);
assert(s->base);
00062
00063
00064
          s->top = -1;
return s;
00065
00066 }
```

6.11.2.2 void stack_destruct (stack_t * s)

Free a stack.

Parameters

```
s a stack
```

Definition at line 73 of file stack.c.

6.11.2.3 int stack_isempty ($stack_t * s$)

Determinate the emptiness of a stack.

Parameters

```
s stack
```

Returns

1 if the stack s is empty, 0 otherwise.

Definition at line 17 of file stack.c.

```
6.11.2.4 void* stack_pop ( stack_t * s )
```

Pop out an element from the stack s.

Parameters

```
s stack
```

Returns

an element

Definition at line 44 of file stack.c.

```
00044 {
00045 if (stack_isempty(s)) return NULL;
00046 s->top--;
00047 return s->base[s->top + 1];
```

Here is the call graph for this function:



```
6.11.2.5 void stack_push ( stack_t * s, void * e )
```

Push an element e into the stack s.

Parameters

s	stack
е	element which be pushed

Definition at line 27 of file stack.c.

6.12 stack.h

Index

adj_list	vertex, 5
gvertex_t, 6	graphs.c, 9
adj_list_len	bfs, 10
gvertex_t, 6	graph_add_edge, 11
	graph_adj_list_print, 12
base	graph_clone, 13
queue_t, 7	graph_create, 13
stack_t, 8	graph_delete_adj_ele, 14
bfs	graph_destruct, 14
graphs.c, 10	graph_dot_output, 15
graphs.h, 22	graph search dot output, 15
3 -1 ,	info_destruct, 16
count	is_adj, 16
queue t, 7	graphs.h, 20
44040 <u>-</u> 1, /	
DIGRAPH	bfs, 22
graphs.h, 22	DIGRAPH, 22
deg_t, 3	graph_add_edge, 23
in, 3	graph_adj_list_print, 24
	graph_clone, 24
out, 3	graph_create, 25
degree	graph_delete_adj_ele, <mark>26</mark>
gvertex_t, 6	graph_destruct, 26
dist	graph_dot_output, 27
info_t, 6	graph_search_dot_output, 27
	gtype_t, 22
front	info_destruct, 28
queue_t, 7	is_adj, 28
	UNDIGRAPH, 22
graph_add_edge	gtype_t
graphs.c, 11	graphs.h, 22
graphs.h, 23	gvertex_t, 5
graph_adj_list_print	adj_list, 6
graphs.c, 12	adj_list, 6
graphs.h, 24	- - ·
graph_clone	degree, 6
graphs.c, 13	in
graphs.h, 24	in
graph_create	deg_t, 3
graphs.c, 13	info_destruct
graphs.h, 25	graphs.c, 16
graph_delete_adj_ele	graphs.h, 28
- ·	info_t, 6
graphs.c, 14	dist, 6
graphs.h, 26	pred, 6
graph_destruct	src, 6
graphs.c, 14	is_adj
graphs.h, 26	graphs.c, 16
graph_dot_output	graphs.h, 28
graphs.c, 15	
graphs.h, 27	m
graph_search_dot_output	graph_t, 4
graphs.c, 15	max_size
graphs.h, 27	queue_t, 7
graph_t, 4	mem_size
m, 4	stack_t, 8
n, 4	otaon_t, o
type, 4	n
-J -J -J -J -J -J -J -J	••

48 INDEX

	graph_t, 4	stack_pop, 44 stack_push, 45
out		stack_create
	deg_t, 3	stack.c, 39
		stack.h, 43
pred		stack_destruct
	info_t, 6	stack.c, 40
~	10.0.00	stack.h, 44
quet	Ie.C, 29	stack_isempty
	queue_create, 30	stack.c, 40
	queue_dequeue, 31	stack.h, 44
	queue_destruct, 31	stack_pop
	queue_enqueue, 32	stack.c, 40
	queue_isempty, 32	stack.h, 44
a	queue_isfull, 33	stack_push
quet	ue.h, 34	stack.c, 41
	queue_create, 35 queue_dequeue, 36	stack.h, 45
	• - •	stack_t, 8
	queue_destruct, 36	base, 8
	queue_enqueue, 37 queue_isempty, 37	mem_size, 8
		top, 8
auai	queue_isfull, 38	width, 9
quet	ue_create	.
	queue.c, 30 queue.h, 35	top
anai	ue_dequeue	stack_t, 8
quet	queue.c, 31	type
	queue.h, 36	graph_t, 4
anai	ue_destruct	UNDIGRAPH
quet	queue.c, 31	graphs.h, 22
	queue.h, 36	graphon, ZZ
ופווח	ue_enqueue	vertex
quet	queue.c, 32	graph_t, 5
	queue.h, 37	• . –
allei	ue_isempty	width
quot	queue.c, 32	queue_t, 8
	queue.h, 37	stack_t, 9
anei	ue_isfull	
90.00	queue.c, 33	
	queue.h, 38	
auei	ue t, 7	
9	base, 7	
	count, 7	
	front, 7	
	max_size, 7	
	width, 8	
src		
	info_t, 6	
stacl	K.C, 38	
	stack_create, 39	
	stack_destruct, 40	
	stack_isempty, 40	
	stack_pop, 40	
	stack_push, 41	
stacl	κ.h, 42	
	stack_create, 43	
	stack_destruct, 44	
	stack isempty, 44	