

Programowanie imperatywne

dr inż. Piotr Szwed
Katedra Informatyki Stosowanej
C2, pok. 403

e-mail: pszwed@agh.edu.pl

<http://home.agh.edu.pl/~pszwed/>

Aktualizacja: 17.05.2020

12. Wybrane implementacje struktur danych i algorytmów

Plan

- Lista dwukierunkowa
- Sortowanie listy (quicksort)
- Stos
- Permutacje – wskaźnik do funkcji
- Permutacje – iterator
- Graf – macierz sąsiedztwa i zbiór krawędzi
- Algorytm Kruskala (minimalne drzewo rozpinające)

Lista dwukierunkowa

Lista dwukierunkowa

```
typedef struct tagDbListElement
{
    struct tagDbListElement*prev;
    struct tagDbListElement*next;
    int data;
}DbListElement;
```

```
typedef struct tagDbList
{
    DbListElement*head;
    DbListElement*tail;
    int size;
}DbList;
```

Lista dwukierunkowa

```
// Funkcja tworzy listę - alokuje pamięć  
// dla struktury i inicjuje pola
```

```
DbList*dbl_create(){  
    DbList*list=malloc(sizeof(DbList));  
    list->head=0;  
    list->tail=0;  
    list->size=0;  
    return list;  
}
```

Lista dwukierunkowa

// Dodawanie danych do listy na początku

```
void dbl_push_front(DbList*list, int data)
{
    DbListElement*element = malloc(sizeof(DbListElement));
    element->prev=0;
    element->next=list->head;
    element->data=data;
    if(list->head!=0){
        list->head->prev=element;
        list->head=element;
    }else{
        list->head=list->tail=element;
    }
    list->size++;
}
```

Lista dwukierunkowa

// Dodawanie danych do listy na końcu

```
void dbl_push_back(DbList*list, int data)
{
    DbListElement*element = malloc(sizeof(DbListElement));
    element->prev=list->tail;
    element->next=0;
    element->data=data;
    if(list->tail!=0){
        list->tail->next=element;
        list->tail=element;
    }else{
        list->head=list->tail=element;
    }
    list->size++;
}
```


Lista dwukierunkowa

// Usuwanie pierwszego elementu

```
void dbl_delete_front(DbList*list){
    DbListElement*toDelete;
    if(list->head==0)return;
    toDelete = list->head;
    list->head=list->head->next;
    if(list->head==0)list->tail=0;
    else list->head->prev=0;
    free(toDelete);
    list->size--;
}
```

Lista dwukierunkowa

```
// Zwalnianie całej listy
void dbl_free(DbList*list){
    while(list->head){
        dbl_delete_front(list);
    }
    printf("\nTRACE: stan listy %p %p %d\n",
        list->head, list->tail, list->size);
    free(list); // usuwamy też strukturę list
}

// Wypisanie zawartości listy,
// iteracja po elementach listy w przód
void dbl_dump(const DbList*list){
    DbListElement*i;
    for(i=list->head; i!=0; i=i->next){
        printf("%d ",i->data);
    }
    printf("\n");
}
```

Lista dwukierunkowa

```
// Wypisanie zawartości listy,  
// iteracja po elementach listy w tył  
void dbl_rdump(const DbList*list){  
    DbListElement*i;  
    for(i=list->tail; i!=0; i=i->prev){  
        printf("%d ",i->data);  
    }  
    printf("\n");  
}  
  
// Iteracja po elementach listy  
// i wywołanie funkcji dla każdego elementu  
void dbl_for_each(const DbList*list, void (*f)(int*)){  
    DbListElement*i;  
    for(i=list->head; i!=0; i=i->next){  
        f(&i->data);  
    }  
}
```

Lista dwukierunkowa

```
// Wypisanie zawartości listy,  
// iteracja po elementach listy w tył  
void dbl_rdump(const DbList*list){  
    DbListElement*i;  
    for(i=list->tail; i!=0; i=i->prev){  
        printf("%d ",i->data);  
    }  
    printf("\n");  
}  
  
// Iteracja po elementach listy  
// i wywołanie funkcji dla każdego elementu  
void dbl_for_each(const DbList*list, void (*f)(int*)){  
    DbListElement*i;  
    for(i=list->head; i!=0; i=i->next){  
        f(&i->data);  
    }  
}
```

Lista dwukierunkowa

```
void odd_print_inc(int*d){
    if(*d%2==1){
        printf("%d ",*d);
        (*d)++;
    }
}

int main(){
    DbList *list = dbl_create();
    for(int i=0;i<15;i++){
        dbl_push_front(list,random()%100);
    }
    dbl_dump(list);
    printf("\n\n");
    dbl_for_each(list,odd_print_inc);
    printf("\n\n");
    dbl_dump(list);
    dbl_free(list);
}
```

Funkcja będzie wołana dla każdego elementu. W przypadku napotkania liczby nieparzystej: wydrukuj i zwiększ o 1

63 59 90 27 62 21 49 92 86 35 93 15 77 86 83

63 59 27 21 49 35 93 15 77 83

84 86 78 16 94 36 86 92 50 22 62 28 90 60 64

TRACE: stan listy 0x0 0x0

Sortowanie listy dwukierunkowej

Quicksort

Zaimplementujemy algorytm *quicksort*

[<https://en.wikipedia.org/wiki/Quicksort>] dla listy. Dla przypomnienia:

- wyszukiwany jest punkt podziału, taki że elementy na lewo są mniejsze, a na prawo większe,
- następnie algorytm jest wywoływany rekurencyjnie dla części lewej i prawej.

Założenia:

- Nie przestawiamy elementów listy, ale ich wartości. Lista ma niezmienną strukturę podczas sortowania
- Zamiast indeksów elementów tablicy – stosujemy wskaźniki do elementów listy
- Wyszukiwanie elementu centralnego – algorytm Lomuto

Quicksort

```
// wymiana elementów
// struktura listy nie zmienia się
static void _dbl_list_swap(DbListElement*a, DbListElement*b){
    int tmp = a->data;
    a->data = b->data;
    b->data = tmp;
}

// Algorytm Lomuto poszukiwania miejsca podziału
static DbListElement* _dbl_partition(DbListElement*p, DbListElement*r){
    int x = r->data;
    DbListElement* i=p;
    for(DbListElement* j=p; j!=r; j=j->next){
        if(j->data < x){
            _dbl_list_swap(i, j);
            i=i->next;
        }
    }
    _dbl_list_swap(i, r);
    return i;
}
```

Użycie **static**: funkcje nie będą widoczne z zewnątrz podczas konsolidacji.
Często prywatne funkcje są specjalnie oznaczane, np. podkreśleniem...

Quicksort

```
// Rekurencyjna implementacja quicksort
// zamiast q-1 mamy q->prev
// zamiast q+1 mamy q->next
static void _dbl_quicksort(DbListElement*p, DbListElement*r){
    if(p!=r){
        DbListElement* q=_dbl_partition(p, r);
        if(p!=q)_dbl_quicksort(p, q->prev);
        if(q!=r)_dbl_quicksort(q->next, r);
    }
}

void dbl_sort(DbList*list){
    _dbl_quicksort(list->head,list->tail);
}
```

Widoczną dla końcowego użytkownika funkcją jest `dbl_sort()`

Quicksort

```
int main(){
    DbList *list = dbl_create();
    for(int i=0;i<20;i++){
        dbl_push_front(list,random()%100);
    }
    dbl_dump(list);
    dbl_sort(list);
    dbl_dump(list);
    dbl_free(list);
}
```

```
36 72 26 40 26 63 59 90 27 62 21 49 92 86 35 93 15 77 86 83
15 21 26 26 27 35 36 40 49 59 62 63 72 77 83 86 86 90 92 93
```

```
TRACE: stan listy 0x0 0x0 0
```

Stos

Stos

Zaimplementujemy stos ogólnego zastosowania, czyli stos, który może przechowywać dane różnych typów (w tej implementacji takich samych).

- Pamięć dla danych będzie alokowana na stercie
- Zaimplementujemy pojemność (`capacity`) i licznik elementów (`cnt`)
- Musimy znać rozmiar elementu (`item_size`).

```
typedef struct {  
    void*stack_data;  
    size_t item_size;  
    size_t capacity;  
    size_t cnt;  
  
    int max_depth; // to do testów  
}stack;
```

Stos – plik nagłówkowy

```
#ifndef STACK_H
#define STACK_H

#ifdef __cplusplus
extern "C" {
#endif

#include <stdlib.h>

typedef struct {
    void*stack_data;
    size_t item_size;
    size_t capacity;
    size_t cnt;

    int max_depth;
}stack;

stack*stack_create(size_t item_size, size_t init_capacity);
void stack_free(stack*s);
int stack_is_empty(stack*s);
void stack_push(stack*s, const void*item);
int stack_pop(stack*s, void*item);

#ifdef __cplusplus
}
#endif

#endif /* STACK_H */
```

Typowy plik nagłówkowy zawiera:

- Definicje typów
- Prototypy funkcji
- Zabezpieczenia przed wielokrotnym włączaniem
- Dyrektywy informujące kompilator C++ o tym, że funkcje modułu są skompilowane kompilatorem C.

```
#ifdef __cplusplus
extern "C" {
#endif
```

Stos

```
stack* stack_create(size_t item_size, size_t init_capacity){
    stack *s = malloc(sizeof(stack));
    s->item_size = item_size;
    s->capacity = init_capacity;
    if(s->capacity){
        s->stack_data=malloc(s->item_size*s->capacity);
    }
    s->cnt=0;
    s->max_depth=0;
    return s;
}

void stack_free(stack*s){
    if(s->stack_data)free(s->stack_data);
    free(s);
}

int stack_is_empty(stack*s){
    return s->cnt==0;
}
```

Stos

```
void stack_push(stack*s, const void*item){
    if(s->capacity==s->cnt){
        s->capacity = (s->capacity==0?20:s->capacity*2);
        s->stack_data=realloc(s->stack_data, s->capacity*s->item_size);
        printf("increasing stack capacity to %d\n",s->capacity);
    }
    memcpy(s->stack_data+(s->cnt++)*s->item_size,item,s->item_size);
    if(s->cnt>s->max_depth){
        s->max_depth=s->cnt;
    }
}

int stack_pop(stack*s, void*item){
    if(s->cnt==0)return 0;
    memcpy(item, s->stack_data + (--(s->cnt))*s->item_size,s->item_size);
    return 1;
}
```

Instrukcje memcpy() są trochę nieczytelne... obliczamy adres
*początek_bloku + licznik * rozmiar_elementu*

Stos

```
int main(){
    stack*sint = stack_create(sizeof(int), 10);
    stack*sstr = stack_create(32, 0);

    for(int i=0;i<10;i++){
        char buf[32];
        stack_push(sint,&i);
        sprintf(buf,"Item%d",i);
        stack_push(sstr,buf);
    }
    while(!stack_is_empty(sint)){
        int k;
        stack_pop(sint,&k);
        printf("%d ",k);
    }
    printf("\n");
    while(!stack_is_empty(sstr)){
        char buf[32];
        stack_pop(sstr,buf);
        printf("%s ",buf);
    }
    stack_free(sint);
    stack_free(sstr);
}
```

Równocześnie używamy dwóch stosów, na dane różnej wielkości – 4 bajty (int) i 32 bajty.

Increasing... Raczej funkcje nie powinny wypisywać komunikatów na stdin. Tu dla pokazania, że sstr powiększa pojemność.

increasing stack capacity to 20

9 8 7 6 5 4 3 2 1 0

Item#9 Item#8 Item#7 Item#6 Item#5 Item#4 Item#3 Item#2 Item#1 Item#

Jak wykorzystać stos?

Na przykład, aby zabezpieczyć się przed ograniczeniami stosu wywołań funkcji.

W przypadku pesymistycznym (np. odwrotne uporządkowanie) rekurencyjna implementacja algorytmu quicksort umieści na stosie n ramek funkcji. Dla $n \approx 33000$ stos przepełni się.

```
int main(){
    DbList *list = dbl_create();

    for(int i=0;i<33000;i++){
        dbl_push_front(list,i);
    }
    dbl_dump(list);
    dbl_sort(list);
    dbl_dump(list);
    dbl_free(list);
}
```

Zmieniamy implementację dbl_sort

```
void _dbl_stack_quicksort(DbListElement*p, DbListElement*r){
    stack*s = stack_create(sizeof(DbListElement*),100);
    stack_push(s, &p);
    stack_push(s, &r);
    while(!stack_is_empty(s)){

        stack_pop(s,&r);
        stack_pop(s,&p);
        DbListElement*q=_dbl_partition( p, r);
        if(p!=q && p!=q->prev){
            stack_push(s, &p);
            stack_push(s, &q->prev);
        }
        if(q!=r && q->next!=r){
            stack_push(s,&q->next);
            stack_push(s, &r);
        }
    }
    stack_free(s);
}

void dbl_sort(DbList*list){
    // _dbl_quicksort(list->head,list->tail);
    _dbl_stack_quicksort(list->head,list->tail);
}
```

Teraz algorytm zadziała
nawet dla 1_000_000
elementów (27 sekund)

Permutacje

Rekurencyjna implementacja (znaleziona w sieci)

```
void swap_ch(char *a, char*b){
    int tmp=*b;
    *b=*a;
    *a=tmp;
}

void permute(char* a, int l, int r)
{
    int i;
    if (l == r)
        printf("%s\n", a);
    else {
        for (i = l; i <= r; i++) {
            swap_ch((a + l), (a + i));
            permute(a, l + 1, r);
            swap_ch((a + l), (a + i)); // backtrack
        }
    }
}

int main(){
    char t[]="ABCD";
    permute(t, 0, strlen(t)-1);
}
```

ABCD
ABDC
ACBD
ACDB
ADCB
ADBC
BACD
BADC
BCAD
BCDA
BDCA
BDAC
CBAD
CBDA
CABD
CADB
CDAB
CDBA
DBCA
DBAC
DCBA
DCAB
DACB
DABC

Jak „skonsumować” permutację

```
void c_permute(char* a, int l, int r, void (*consumer)(const char*))
{
    int i;
    if (l == r)
        //printf("%s\n", a);
        consumer(a);
    else {
        for (i = l; i <= r; i++) {
            swap_ch((a + l), (a + i));
            c_permute(a, l + 1, r, consumer);
            swap_ch((a + l), (a + i)); // backtrack
        }
    }
}

static int cnt=0;
void permutations_counter(const char*a){
    cnt++;
}

int main(){
    char t[]="ABCDEFGHGIJK";
    c_permute(t,0,strlen(t)-1,permutations_counter);
    printf("#permutations = %d ",cnt);
}
```

Do funkcji `c_permute()` przekazujemy wskaźnik do funkcji, która w jakiś sposób wykorzysta (skonsumuje) permutację.

W naszym przypadku policzy je:
#permutations = 39916800

Użycie zmiennej globalnej nie jest zalecane.

Zamiast modyfikacji zmiennej globalnej

```
void ca_permute(char* a, int l, int r, void (*consumer)(const char*,void*),void*arg)
{
    int i;
    if (l == r)
        //printf("%s\n", a);
        consumer(a,arg);
    else {
        for (i = l; i <= r; i++) {
            swap_ch((a + l), (a + i));
            ca_permute(a, l + 1, r,consumer,arg);
            swap_ch((a + l), (a + i)); // backtrack
        }
    }
}

void permutations_counter2(const char*a,void*arg){
    (*(int*)arg)++;
}

int main(){
    char t[]="ABCDEFGHIIJK";
    int cnt=0;
    ca_permute(t,0,strlen(t)-1,permutations_counter2,&cnt);
    printf("#permutations = %d ",cnt);
}
```

Zadeklarujemy funkcję konsumenta tak, aby dodatkowo mogła otrzymywać argument typu void*.

Przy wywołaniu prześlemy adres zadeklarowanej lokalnie zmiennej cnt służącej do zliczania permutacji.

#permutations = 39916800

Permutacje z własnym stosem

```
void stack_permute(char* a, int r){
    stack*s = stack_create(sizeof(int),10);
    //first
    for(int i=0;i<=r;i++){
        stack_push(s,&i);
        stack_push(s,&i);
    }
    printf("%s\n", a);
    int end =0;
    while(!end){
        // next
        int i,l;
        for(;;){
            stack_pop(s,&i);
            stack_pop(s,&l);
            swap_ch((a + l), (a + i)); // backtrack
            if(i<r)break;
            if(stack_is_empty(s)){end=1;break;}
        }
        i++;
        for(;l<=r;l++,i=1){
            swap_ch((a + l), (a + i));
            stack_push(s,&l);
            stack_push(s,&i);
        }
        printf("%s\n", a);
    }
    stack_free(s);
}
```

- **Fragment //first**
Wyznacza pierwszą permutację i przygotowuje dane na stosie
- **Fragment //next**
 1. Wpierw cofa się zwalniając stos
 2. Jeżeli stos pusty – koniec.
 3. Dalej idzie w kierunku wyznaczenia kolejnej permutacji

ABCD
ABDC
ACBD
ACDB
ADCB
ADBC
BACD
BADC
BCAD
BCDA
BDCA
BDAC
CBAD
CBDA
CABD
CADB
CDAB
CDBA
DBCA
DBAC
DCBA
DCAB
DACB
DABC

Iteracja po permutacjach

Czy proces wyznaczania kolejnych permutacji można zamienić na abstrakcyjny typ danych:

- Strukturę przechowującą stan
- Zestaw funkcji pozwalających na iterację
 - `pi_init()` – inicjuje wyznaczanie permutacji
 - `pi_next()` – wyznacza następną permutację
 - `pi_at_end()` – czy wyznaczono już wszystkie permutacje
 - `pi_get()` – zwraca bieżącą permutację
 - `pi_close()` – kończy i zwalnia zasoby

Iterator

```
typedef struct{
    stack *s;
    char*tab;
    int r;
    int at_end;
}permutation_iterator;

permutation_iterator*pi_init(const char*text);
int pi_at_end(permutation_iterator*pi);
int pi_next(permutation_iterator*pi);
const char*pi_get(permutation_iterator*pi);
void pi_close(permutation_iterator*pi);
```

```
int main(){
    char t[]="ABCD";
    permutation_iterator* pi = pi_init(t);
    while(!pi_at_end(pi)){
        const char*ptr=pi_get(pi);
        printf("%s\n",ptr);
        pi_next(pi);
    }
    pi_close(pi);
}
```

Iterator (czasem generator) to obiekt, który przechowuje stan iteracji. Pozwala na dostęp do bieżącego elementu i „przesuwa kursor” na następny element. Pozwala na sprawdzenie, czy osiągnięto koniec iteracji.

ACBD
ACDB
ADCB
ADBC
BACD
BADC
BCAD
BCDA
BDCA
BDAC
CBAD
CBDA
CABD
CADB
CDAB
CDBA
DBCA
DBAC
DCBA
DCAB
DACB
DABC

Iterator

```
permutation_iterator*pi_init(const char*text){
    permutation_iterator*pi=malloc(sizeof(permutation_iterator));
    pi->tab = strdup(text);
    pi->r = strlen(text)-1;
    pi->at_end=0;
    pi->s= stack_create(sizeof(int),10);
    ////
    for(int i=0;i<=pi->r;i++){
        stack_push(pi->s,&i);
        stack_push(pi->s,&i);
    }
    ////
    return pi;
}

void pi_close(permutation_iterator*pi){
    free(pi->tab);
    stack_free(pi->s);
    free(pi);
}
```

Iterator

```
int pi_next(permutation_iterator*pi){
    int i,l;
    for(;;){
        stack_pop(pi->s,&i);
        stack_pop(pi->s,&l);
        swap_ch((pi->tab + l), (pi->tab + i)); // backtrack
        if(i < pi->r)break;
        if(stack_is_empty(pi->s)){
            pi->at_end=1;
            return 0;
        }
    }
    i++;
    for(;l<=pi->r;l++,i=l){
        swap_ch((pi->tab + l), (pi->tab + i));
        stack_push(pi->s,&l);
        stack_push(pi->s,&i);
    }
    return 1;
}
```

Iterator

```
const char*pi_get(permutation_iterator*pi){  
    return pi->tab;  
}  
int pi_at_end(permutation_iterator*pi){  
    return pi->at_end;  
}
```

```
int main(){  
    char t[]="ABCDEFGHIIJK";  
    permutation_iterator* pi = pi_init(t);  
    int cnt=0;  
    while(!pi_at_end(pi)){  
        pi_next(pi);  
        cnt++;  
    }  
    pi_close(pi);  
    printf("#permutations = %d ",cnt);  
}
```

increasing stack capacity to 20
increasing stack capacity to 40
#permutations = 3991680

Grafy

Grafy

Trochę teorii...

Graf $G = (V, E, W)$, gdzie V to zbiór wierzchołków, $E \subset V \times V$ to zbiór krawędzi, $W: E \rightarrow R$ to funkcja przypisująca krawędziom wagi (liczby rzeczywiste).

Graf może być reprezentowany przez macierz sąsiedztwa (ang. adjacency matrix), o wymiarach $n \times n$, gdzie n to liczba wierzchołków. Zera oznaczają brak połączeń, wartości niezerowe to wagi. W przypadku grafu nieskierowanego ta macierz jest symetryczna.

```
void*random_adj_matrix(int n, double density){
    double (*d)[n] = calloc(n*n,sizeof(double));

    for(int i=0;i<n;i++){
        for(int j=i+1;j<n;j++){
            double r = rand()*1.0/RAND_MAX;
            if(r>density)continue;
            d[i][j]=d[j][i]=rand()%100/10.0;
        }
    }
    return d;
}
```

Parametr density to określa próg prawdopodobieństwa. Pomijamy elementy na przekątnej.

Rzeczywista gęstość (liczba niezerowych elementów / n^2) może być różna od density

Grafy

```
void print_matrix(int n, double (*d)[n]){
    double cnt=0;
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            printf("%2.1f ",d[i][j]);
            if(d[i][j]>0)cnt++;
        }
        printf("\n");
    }
    printf("density=%f\n",cnt/(n*n-n));
}
```

```
void adj_to_dot(int n, double (*d)[n]){
    printf("graph g{\n");
    for(int i=0;i<n;i++){
        for(int j=i+1;j<n;j++){
            if(d[i][j]!=0){
                printf("%d -- %d [label = %.1f]\n",i,j,d[i][j]);
            }
        }
    }
    printf("}\n");
}
```

Dodamy dwie funkcje do wypisania zawartości.
Pierwsza, print_matrix() drukuje zawartość tablicy.

Druga, adj_to_dot() generuje opis grafu nieskierowanego w języku dot (pakiet Graphviz [<https://www.graphviz.org/>]).

Grafy – macierz sąsiedztwa

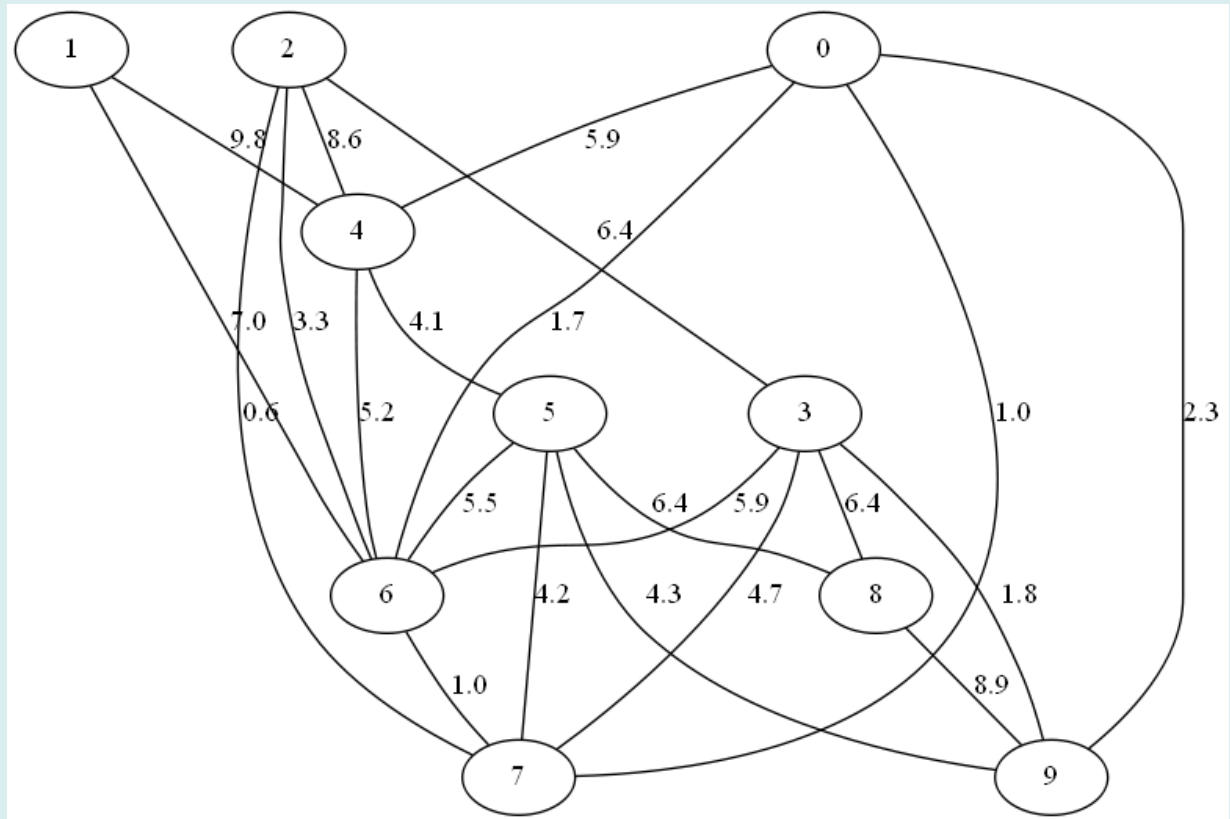
```
int main(){
    int size=10;
    srand(4);
    double (*d)[size] = random_adj_matrix(size,0.5);
    print_matrix(size,d);
    adj_to_dot(size,d);
    free(d);
}
```

Zawartość macierzy

```
0.0 0.0 0.0 0.0 5.9 0.0 1.7 1.0 0.0 2.3
0.0 0.0 0.0 0.0 9.8 0.0 7.0 0.0 0.0 0.0
0.0 0.0 0.0 6.4 8.6 0.0 3.3 0.6 0.0 0.0
0.0 0.0 6.4 0.0 0.0 0.0 5.9 4.7 6.4 1.8
5.9 9.8 8.6 0.0 0.0 4.1 5.2 0.0 0.0 0.0
0.0 0.0 0.0 0.0 4.1 0.0 5.5 4.2 6.4 4.3
1.7 7.0 3.3 5.9 5.2 5.5 0.0 1.0 0.0 0.0
1.0 0.0 0.6 4.7 0.0 4.2 1.0 0.0 0.0 0.0
0.0 0.0 0.0 6.4 0.0 6.4 0.0 0.0 0.0 8.9
2.3 0.0 0.0 1.8 0.0 4.3 0.0 0.0 8.9 0.0
density=0.488889
```


Grafy – opis w dot i wizualizacja

```
graph g{
0 -- 4 [label = 5.9]
0 -- 6 [label = 1.7]
0 -- 7 [label = 1.0]
0 -- 9 [label = 2.3]
1 -- 4 [label = 9.8]
1 -- 6 [label = 7.0]
2 -- 3 [label = 6.4]
2 -- 4 [label = 8.6]
2 -- 6 [label = 3.3]
2 -- 7 [label = 0.6]
3 -- 6 [label = 5.9]
3 -- 7 [label = 4.7]
3 -- 8 [label = 6.4]
3 -- 9 [label = 1.8]
4 -- 5 [label = 4.1]
4 -- 6 [label = 5.2]
5 -- 6 [label = 5.5]
5 -- 7 [label = 4.2]
5 -- 8 [label = 6.4]
5 -- 9 [label = 4.3]
6 -- 7 [label = 1.0]
8 -- 9 [label = 8.9]
}
```



Grafy – lista krawędzi

Grafy mogą także być reprezentowane w postaci listy (tablicy) krawędzi.

```
typedef struct {  
    int first;    // numer pierwszego wierzchołka  
    int second;   // numer drugiego wierzchołka  
    double weight; // waga  
} edge;
```

Liczbę wierzchołków w grafie można po prostu obliczyć:

```
int get_vertex_count(const edge*edges, int n){  
    int max=-1;  
    for(int i=0;i<n;i++){  
        if(edges[i].first>max)max=edges[i].first;  
        if(edges[i].second>max)max=edges[i].second;  
    }  
    return max+1;  
}
```

Grafy

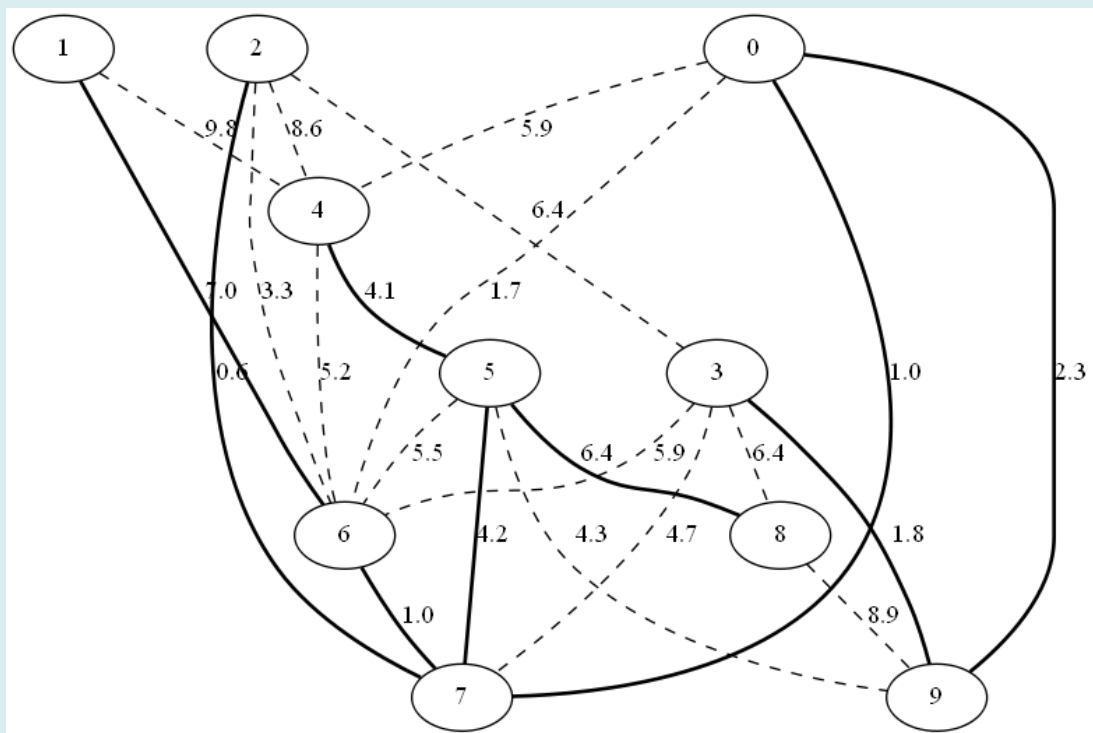
Konwersja macierzy sąsiedztwa do postaci tablicy krawędzi

```
edge* from_matrix(int n, double (*m)[n], int*ecount){
    edge*edges = 0;
    int count=0;
    int capacity=0;
    for(int i=0;i<n;i++){
        for(int j=i+1;j<n;j++){
            if(m[i][j]!=0){
                if(count==capacity){
                    if(capacity==0)capacity=10;
                    else capacity=2*capacity;
                    edges = realloc(edges,capacity*sizeof(edge));
                }
                edges[count].first=i;
                edges[count].second=j;
                edges[count].weight=m[i][j];
                count++;
            }
        }
    }
    *ecount = count;
    return edges;
}
```

Algorytm Kruskala

Algorytm Kruskala służy do wyznaczania minimalnego drzewa rozpinającego (ang. MST) grafu, czyli takiego podgrafu będącego drzewem, które minimalizuje sumę wag krawędzi.

- Wszystkie wierzchołki muszą być połączone w jeden spójny podgraf
- Podgraf musi być drzewem (nie może zawierać cykli)
- Suma wag krawędzi - minimalna



MST oznaczone
pogrubionymi
krawędziami

Algorytm Kruskala

1. Posortuj krawędzie E według wag
2. Utwórz n komponentów $C_i = \{v_i\}$ dla każdego wierzchołka $v_i \in V$
3. Dla kolejnych krawędzi $e = (u, v, w)$ w tablicy E :
 - $c_1 = FIND(u)$
 - $c_2 = FIND(v)$
 - Jeżeli $c_1 \neq c_2$:
 - dodaj: $MST \leftarrow MST \cup e$
 - połącz komponenty: $UNION(c_1, c_2)$

- Uporządkowanie krawędzi gwarantuje, że wpierw analizowane są krawędzie o najmniejszych wagach.
- Komponenty, to poddrzewa. Warunek $c_1 \neq c_2$ gwarantuje, że nie wprowadzimy krawędzi, której oba wierzchołki należą do tego samego komponentu (czyli nie wprowadzamy cyklu).
- Konieczna implementacja dwóch operacji:
 - $FIND(u)$ – zwraca komponent, do którego należy wierzchołek u
 - $UNION(c_1, c_2)$ – łączy ze sobą komponenty c_1 i c_2

Union-find

Union-find to abstrakcyjny typ danych z dwiema głównymi funkcjami:

- `union()` – łączy dwa komponenty
- `find()` – zwraca numer komponentu, do którego należy wierzchołek

```
typedef struct {  
    int n;           // number of nodes  
    int c_count;     // number of components  
    int*n2c;         // node to component  
    int*c_nodes;     // component nodes (linked list)  
    int*c_start;     // start address in c_nodes  
    int*c_sizes;     // component sizes  
}union_find;  
  
union_find*uf_create(int n);  
void uf_free(union_find*uf);  
int uf_find(union_find*uf, int node);  
int uf_union(union_find*uf, int c1, int c2);  
  
void uf_print(const union_find*uf);
```

Union-find

Przykład: union_find dla 6 wierzchołków oraz 2 komponentów: c_1 o indeksie 1 oraz c_2 o indeksie 2

n2c

1	1	2	1	2	1
---	---	---	---	---	---

n2c – tablica zawiera numery komponentów przypisanych wierzchołkom

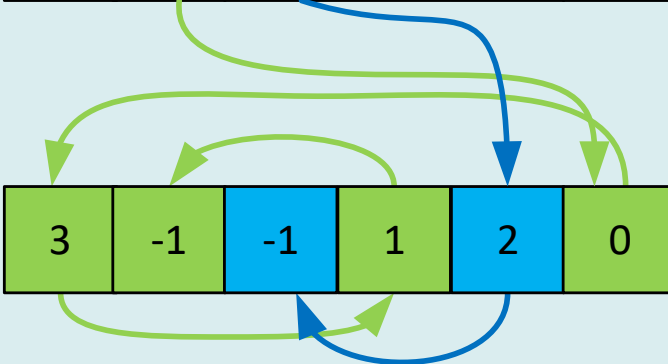
c_start

-1	5	4	-1	-1	-1
----	---	---	----	----	----

c_start: lista wierzchołków c_1 rozpoczyna się od wierzchołka 5, natomiast c_2 od 4

c_nodes

3	-1	-1	1	2	0
---	----	----	---	---	---



c_nodes: listy wierzchołków to $5 \rightarrow [0, 3, 1]$ oraz $4 \rightarrow [2]$

c_sizes

0	4	2	0	0	0
---	---	---	---	---	---

c_sizes – komponent o numerze 1 (c_1) ma 4 wierzchołki, komponent o numerze 2 (c_2) ma 2 wierzchołki

Union-find

```
union_find*uf_create(int n){
    union_find*uf = malloc(sizeof(union_find));
    uf->n=n;
    uf->c_count=n;
    uf->n2c = malloc(n*sizeof(int));
    for(int i=0;i<n;i++)uf->n2c[i]=i;

    uf->c_nodes = malloc(n*sizeof(int));
    for(int i=0;i<n;i++)uf->c_nodes[i]=-1;

    uf->c_start = malloc(n*sizeof(int));
    for(int i=0;i<n;i++)uf->c_start[i]=i;

    uf->c_sizes = malloc(n*sizeof(int));
    for(int i=0;i<n;i++)uf->c_sizes[i]=1;
    return uf;
}
```


Union-find

```
void uf_free(union_find*uf){
    free(uf->n2c);
    free(uf->c_nodes);
    free(uf->c_start);
    free(uf->c_sizes);
    free(uf);
}

int uf_find(union_find*uf, int node){
    return uf->n2c[node];
}
```

```

int uf_union(union_find*uf,int c1, int c2){
    if(uf->c_start[c1]<0)return 0;
    if(uf->c_start[c2]<0)return 0;
    if(uf->c_sizes[c1]==0)return 0;
    if(uf->c_sizes[c2]==0)return 0;

    if(uf->c_sizes[c1]<uf->c_sizes[c2]){
        int tmp = c1;
        c1=c2;
        c2=tmp;
    }
    // c1 is larger -> update c2
    int last=-1;
    for(int i=uf->c_start[c2];i!=-1;i=uf->c_nodes[i]){
        uf->n2c[i]=c1;
        last=i;
    }
    uf->c_nodes[last] = uf->c_start[c1];
    uf->c_start[c1] = uf->c_start[c2];
    uf->c_start[c2]=-1;
    uf->c_sizes[c1]+=uf->c_sizes[c2];
    uf->c_sizes[c2]=0;
    uf->c_count--;
    return 1;
}

```

Podczas operacji union() przepinamy wskaźniki oraz uaktualniamy n2c.

Zmieniamy numer komponentu o mniejszej liczbie wierzchołków.

Tutaj: c2 jest dołączany do c1

```
typedef struct{
    int *data;
    int size;
    int capacity;
}int_vect;

int_vect*iv_create(){
    int_vect*iv=malloc(sizeof(iv));
    iv->data=0;
    iv->size=0;
    iv->capacity=0;
    return iv;
}

void iv_free(int_vect*iv){
    if(iv->data)free(iv->data);
    free(iv);
}

void iv_append(int_vect*iv,int value){
    if(iv->size==iv->capacity){
        if(iv->capacity==0)iv->capacity=10;
        else iv->capacity*=2;
        iv->data = realloc(iv->data,(iv->capacity)*sizeof(int));
    }
    iv->data[iv->size]=value;
    iv->size++;
}
```

Numer krawędzi MST będą przechowywane w tablicy.

int_vect to automatycznie powiększająca się tablica liczb całkowitych, dla której pamięć jest alokowana na stacku.

Krawędzie muszą zostać posortowane

```
int cmp_weight(const void*a, const void*b){
    const edge*ea=(const edge*)a;
    const edge*eb=(const edge*)b;
    if(ea->weight<eb->weight)return -1;
    if(ea->weight==eb->weight)return 0;
    return 1;
}

int_vect* kruskal(edge*edges, int n) {
    qsort(edges, n, sizeof(edge), cmp_weight);
    ...
}
```

Funkcja kruskal()

```
int_vect* kruskal(edge*edges, int n){
    qsort(edges,n,sizeof(edge),cmp_weight);
    int nodes = get_vertex_count(edges,n);
    union_find*uf = uf_create(nodes);
    int_vect*iv=iv_create();

    for(int i=0;i<n;i++){
        if(uf->c_count==1)break;
        int c1=uf_find(uf,edges[i].first);
        int c2=uf_find(uf,edges[i].second);

        if(c1==c2)continue;
        iv_append(iv,i);
        uf_union(uf,c1,c2);
    }
    if(uf->c_count!=1){
        printf("not coherent %d\n",uf->c_count);
    }
    uf_free(uf);
    return iv;
}
```

Wizualizacja w Graphviz

```
void to_dot(edge*edges,int n_edges,int*mst,int n){
    printf("graph g{\n");
    for(int i=0;i<n_edges;i++){
        printf("%d -- %d [label = %.1f",
                edges[i].first,
                edges[i].second,
                edges[i].weight);

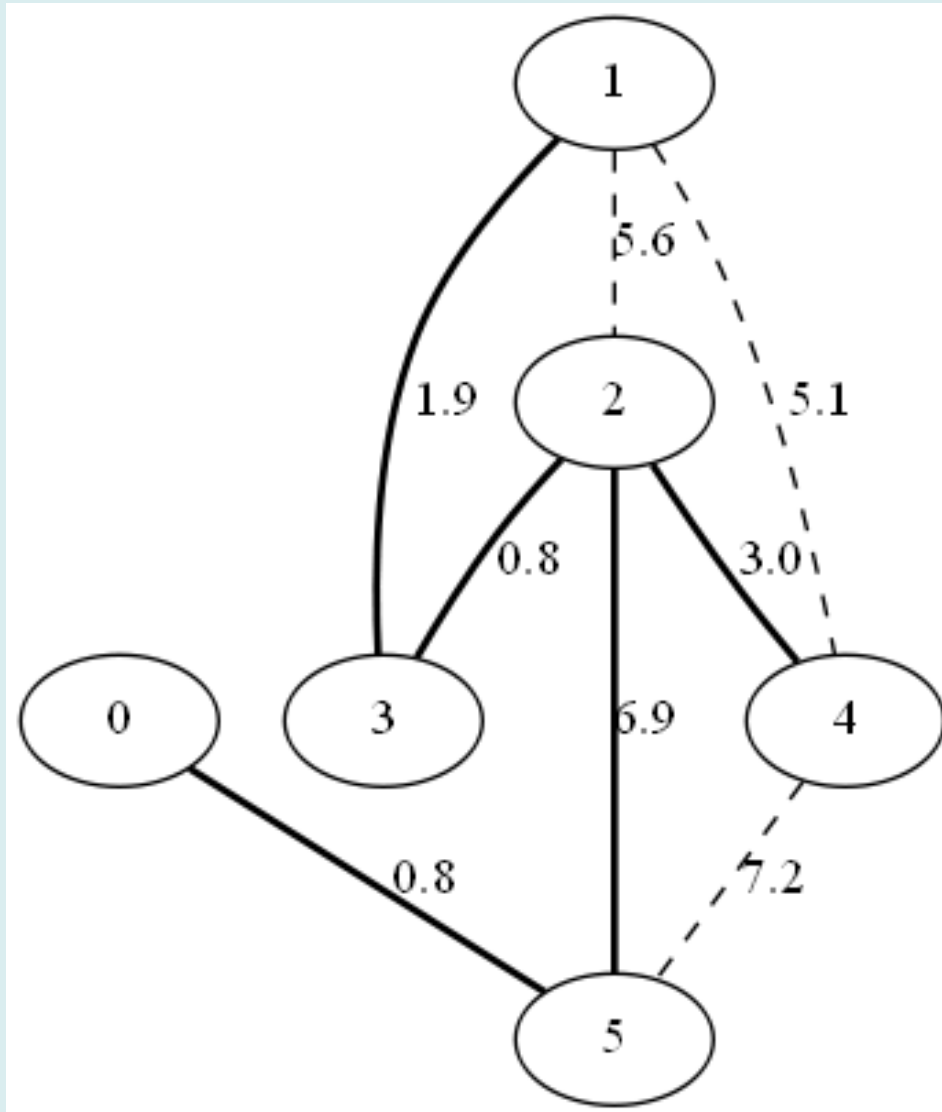
        int found=0;
        for(int j=0;j<n;j++)
            if(i==mst[j]){
                found=1;
                break;
            }
        if(found)printf(" style = bold]\n");
        else printf(" style = dashed]\n");
    }
    printf("}\n");
}
```

main()

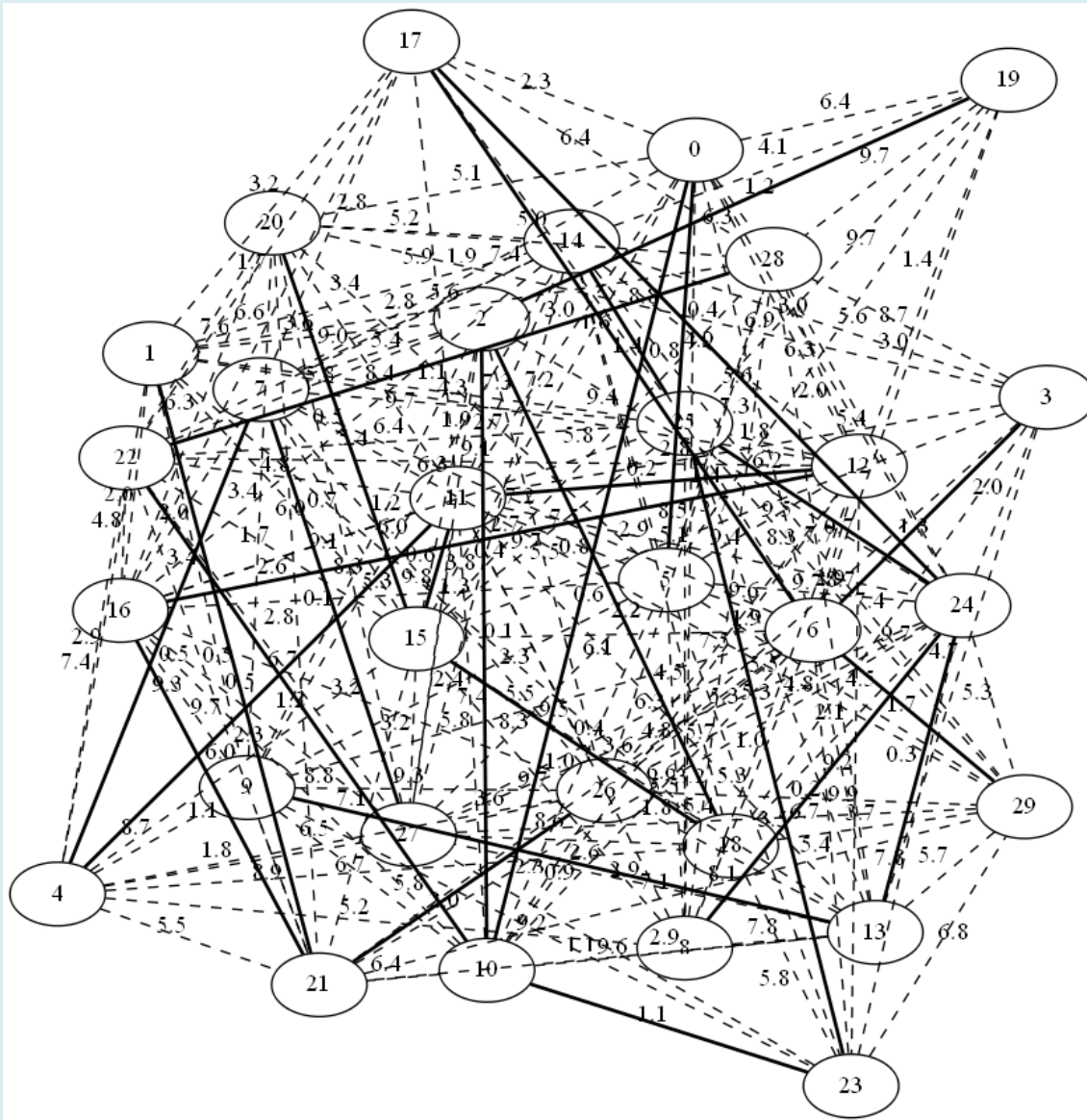
```
int main(){
    int size=20000;
    srand(1);
    clock_t t1=clock();
    double (*d)[size] = random_adj_matrix(size,0.5);
    clock_t t2=clock();
    int n_edges=0;
    edge* edges=from_matrix(size, d,&n_edges);
    clock_t t3 = clock();
    int_vect*iv = kruskal(edges,n_edges);
    clock_t t4 = clock();
    //    to_dot(edges,n_edges,iv->data,iv->size);
    printf("size=%d t1t2=%f t2t3=%f t3t4=%f\n",size,
        (double)(t2-t1)/CLOCKS_PER_SEC,
        (double)(t3-t2)/CLOCKS_PER_SEC,
        (double)(t4-t3)/CLOCKS_PER_SEC
    );
    iv_free(iv);
    free(edges);
    free(d);
}
```

size=20000
t1t2=10.016000
t2t3=8.624000
t3t4=9.609000

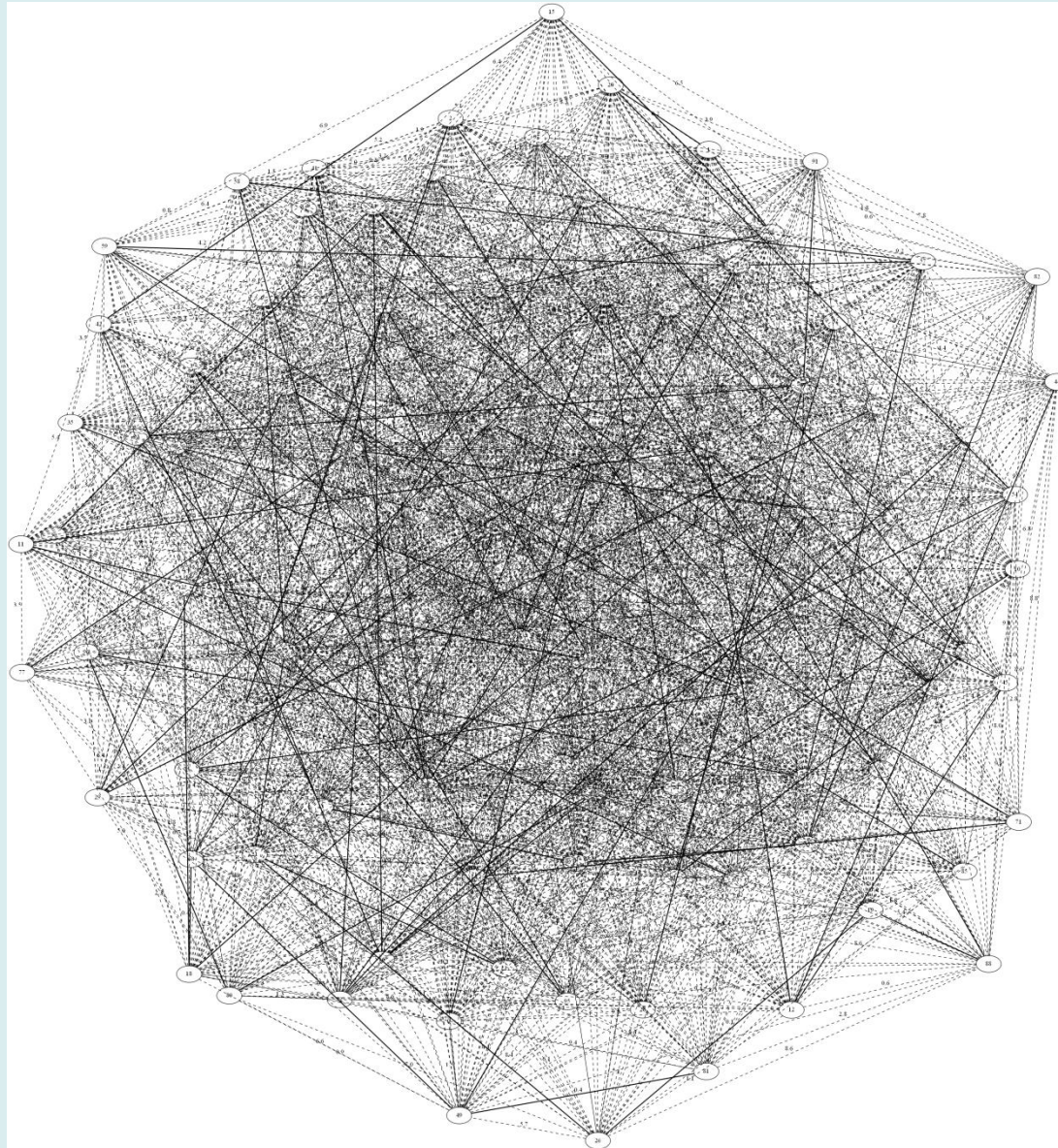
MST n=6



MST n=30



MST $n=100$



Do zapamiętania

- Programy często składają się z niewielkich komponentów połączonych w użyteczną całość
- Najczęściej komponenty mają postać **struktury** oraz **funkcji** działających na tej strukturze (tworzenie, zwalnianie, dodawanie danych, usuwanie, itd.). W obiektowych językach programowania odpowiednikiem struktury i zbioru funkcji są klasy.
- Listy, drzewa, powinny być implementowane z użyciem dwóch struktur: pierwsza to element przechowujący dane (element listy, węzeł drzewa), drugi to struktura (`List`, `Tree`) zapewniająca interfejs dostępu oraz przechowująca wskaźniki do głównych/początkowych elementów.
- Język C ma słabe mechanizmy organizacji kodu. Programista powinien zadbać o unikalne nazwy funkcji, stąd propozycja stosowania przedrostków dla grup funkcji działających na jednej strukturze.
- Funkcje „prywatne” w miarę możliwości ukrywamy stosując modyfikator `static`. Prototypy funkcji „publicznych” powinny znaleźć się w pliku nagłówkowym razem z deklaracją struktury.