WDI Projekt Semestralny 2021L

Kacper Cybiński

27 czerwca 2021

Streszczenie

Został napisany program znajdujący *silnie spójne składowe* grafu zorientowanego. Został on zaimplementowany z użyciem wszystkich typów struktur, których się uczyliśmy w tym roku, tj. list, drzew i grafów. Jego złożoność to $O(|V| + |E|)$ z racji oparcia go na przeszukiwaniu DFS.

1 Wstęp

Jako swój projekt semestralny wybrałem zadanie **nr.4**. Temat to:

Znajdywanie silnie spójnych składowych grafu zorientowanego.

Algorytm, z którego skorzystałem w napisanym przez siebie programie został przedstawiony na wykładzie prof. Zawadowskiego z dnia 10.05.21 r[2], jest on oparty na Algorytmie Kosaraju [1]. Jego zasada działania została opisana w Rozdziale 3. Kod programu jest załączony wraz z tym raportem, lub może być znaleziony w moim repozytorium na GitHub.

2 Wstęp Teoretyczny

W grafie zorientowanym $G = (V, E)$, gdzie (V, E) jest parą zorientowaną taką, że:

- V - zbiór wierzchołków, będących kolejnymi liczbami całkowitymi od 0 do n,
- E - podzbiór $V \times V$ o którego elemencie $e = (u, v) \in E$ mówimy, że jest *krawędzią* grafu, jeśli dla $u, v \in V$ e zadaje niezwrotną relację binarną, która łączy wierzchołek u z wierzchołkiem v. O takiej krawędzi mówimy, że *prowadzi z u do v*.

Dla tak zdefiniowanego grafu określmy relację $\mapsto$, która dla $u, v \in V$ oznacza, że istnieje ścieżka zadana *krawędziami* grafu G z wierzchołka u do v. Wtedy możemy zdefiniować na tym zbiorze przechodnią i symetryczną relację binarną $\sim$ taką, że:

$$u \sim v \iff u \mapsto v \wedge v \mapsto u$$

Silnie spójnymi składowymi grafu G nazwiemy klasy abstrakcji relacji $\sim$.

Przedstawiając tę ideę w prostych słowach możemy powiedzieć, że *silnie spójną składową* grafu G jest taki jego podgraf, że istnieje w jego obrębie ścieżka w obie strony $\mapsto$ między każdymi dwoma elementami tego grafu.

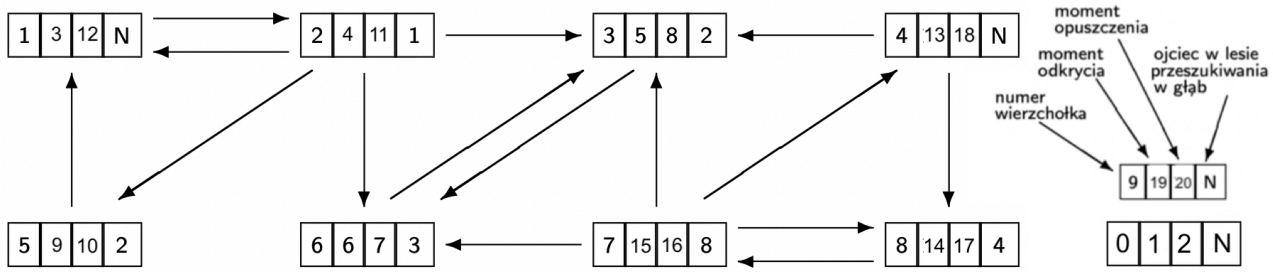
3 Zasada działania algorytmu

Przedstawiony na Rysunku 1 graf został zakodowany w programie dla celów demonstracyjnych. Na jego podstawie zostanie omówiona zasada działania algorytmu.

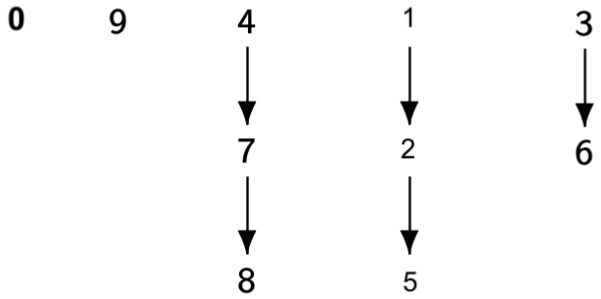
Kroki algorytmu:

1. Przeszukujemy graf G w głąb procedurą DFS w celu obliczenia tablicy f - czasów opuszczenia wierzchołka
2. Obliczamy graf transponowany G^T
3. Przeszukujemy graf G^T wgłąb procedurą DFS z tym, że tym razem w pętli głównej procedury DFS przeszukujemy wierzchołki w porządku malejących wartości poprzednio obliczonego f
4. Wypisujemy wierzchołki drzew lasu przeszukiwania w głąb grafu G^T jako kolejne silnie spójne składowe grafu G.

Z racji oparcia algorytmu na procedurze DFS, wyznacza ona maksymalną złożoność programu jako $O(|V|+|E|)$, tj. złożoność liniową. Wg literatury[1] złożoność liniowa jest minimalną osiągalną złożonością do rozwiązania tego typu problemu.



Rysunek 1: Przykładowy graf $G = (V, E)$ na bazie którego zostało zademonstrowane działanie programu. W programie moment odkrycia jest określany jako d, moment opuszczenia jako f, a ojciec w lesie przeszukiwania wgłąb jako $prev\_node$.



Rysunek 2: Silnie spójne składowe grafu G będące drzewami składającymi się na las powstały z zastosowania algorytmu DFS do grafu transponowanego G^T idąc po wierzchołkach w porządku malejącej wartości f - czasu opuszczenia wierzchołka przy algorytmie DFS do grafu G.

4 Dokumentacja kodu

Każda z funkcji programu została podpisana w komentarzach w kodzie, w tej sekcji raportu będą wymienione co ważniejsze z nich, oraz struktury danych na jakich pracuje program. Program został podzielony na 5 głównych części. Najpierw mamy 3 sekcje pomocniczych funkcji technicznych. Potem dwie sekcje funkcji zasadniczych. Kolejno kodują one:

1. Drzewa
2. Grafy i listy incydencji
3. Listy
4. Algorytm DFS
5. Algorytm do znajdowania *Silnie Spójnych Składowych* (SSS).

4.1 Drzewa

4.1.1 Struktura danych

Struktura *drzewa* została stworzona z elementów opisanych w programie jako `tree`, z zastosowaniem aliasu `key_type` dla typu `int`. Element tree ma konstrukcję:

```
    typedef struct node
{
  key_type key;
  struct node *left;
  struct node *right;
  struct node *up;
} node, *tree;
```

Oznacza to, że każdy z elementów drzewa przechowuje w sobie swoją wartość - `key`, wskaźnik do ojca - `*up`, prawego syna - `*right` i lewego syna - `*left`.

Drzewo jest inicjowane poprzez przypisanie `NULL` elementowi `tree`

4.1.2 Funkcje do obsługi struktury drzewa

Do obsługi struktury drzewa zaimplementowano następujące funkcje:

- `tree new_vertex(key_type a)` - odpowiadające za stworzenie nowego wierzchołka w drzewie
- `void init_BST(tree *r)` - inicjalizujące drzewo
- `void add_vertex_BST(tree *r, tree v)` - dodająca węzeł o podanym kluczu
- `void spacje(int n)` - f.pomocnicza do rysowania drzewa funkcją `void rysuj`
- `void rysuj(tree t, int n)` - funkcja do graficznego przedstawienia drzewa

4.2 Grafy i Listy Incydencji

4.2.1 Struktura danych

Przy konstruowaniu grafu zastosowano alias `typedef enum{white, gray, black} colors` dla listy możliwych kolorów wierzchołków w algorytmie DFS. Struktura *grafu* została stworzona z elementów opisanych w programie jako `adj_list_node_p`, `adj_list_p`, `graph_node_p`, oraz `graph_p`.

- Element `adj_list_node_p` koduje wierzchołek w liście incydencji i ma konstrukcję:

```
typedef struct adj_list_node
{
  int id; // Wartość na wierzchołku
  struct adj_list_node *next;
} adj_list_node, *adj_list_node_p;
```

- Element `adj_list_p` koduje listę incydencji i ma konstrukcję:

```
typedef struct adj_list
{
  adj_list_node_p head; // Głowa listy
} adj_list, *adj_list_p;
```

- Element `graph_node_p` koduje element grafu i ma konstrukcję:

```
typedef struct graph_node
{
  colors color; // Kolor wierzchołka
  int d; // Czas odkrycia wierzchołka
  int f; // Czas opuszczenia wierzchołka
  int prev_node; // Poprzednik na ścieżce w grafie
} graph_node, *graph_node_p;
```

- Element `graph_p` koduje strukturę grafu i ma konstrukcję:

```
typedef struct graph
{
  int V; // Ilość wierzchołków w grafie
  graph_node_p info_v; // Dane wierzchołka
  adj_list_p array; // Lista incydencji
} graph, *graph_p;
```

Oznacza to, że każdy z elementów grafu przechowuje w sobie dane o sobie w elemencie `info_v`(`color`, `d`, `f`, `prev_node`) , listę incydencji - `array`, oraz ilość wierzchołków grafu - `V`.

Inicjalizacja grafu jest zdefiniowana funkcją

```
graph_p create_graph(int V)
{
  graph_p g = malloc(sizeof(graph_p));
  g->V = V;
  g->info_v = malloc(V * sizeof(graph_node));
  g->array = malloc(V * sizeof(adj_list));
  int i;
  for(i = 0; i < V; i++){
    g->array[i].head = NULL;
  }
  return g;
}
```

4.2.2 Funkcje opisujące grafy

- Funkcja, która tworzy nowy element, który następnie będzie dodawany do odpowiedniej listy incydencji

```
adj_list_node_p new_adj_node(int id)
{
  adj_list_node_p new_node = malloc(sizeof(adj_list_node));
  new_node->id = id;
  new_node->next = NULL;
  return new_node;
}
```

- Procedura, która dodaje krawędzie w grafie

```
void add_edge(graph_p g, int src, int dest)
{
  adj_list_node_p new_node = new_adj_node(dest);
  new_node->next = g->array[src].head;
  g->array[src].head = new_node;
}
```

- Procedura przepisująca graf G na graf transponowany G^T

```
void transpose_graph(graph_p g, graph_p g_t)
{
  int v;
  for (v = 0; v < g->V; v++){
    adj_list_node_p act_list = g->array[v].head;
    while (act_list){
      add_edge(g_t, act_list->id, v);
      act_list = act_list->next;
    }
  }
}
```

- Procedury drukujące graf na różne sposoby:
  - `void print_graph(graph_p g)` - drukuje listę incydencji z grafu
  - `void print_graph_time(graph_p g)` - drukowanie listy incydencji wraz z czasami odkrycia i opuszczenia wierzchołków
  - `void print_nodes (graph_p g)` - Wydrukowanie wierzchołków grafu wraz z wartościami prev_node, d, f dla nich

4.3 Listy

4.3.1 Struktura danych

Struktura *listy* została stworzona z elementów opisanych w programie jako `list`, będącym aliasem dla typu `adj_list_node_p`. Element `list` dziedziczy konstrukcję z elementu `adj_list_node_p` opisanego w rozdziale 4.2.1.

Oznacza to, że każdy z elementów listy przechowuje w sobie swoją wartość - `id` i wskaźnik do kolejnego elementu - `*next`.

Lista jest inicjowana poprzez przypisanie `NULL` głowie listy - elementowi `list`.

4.3.2 Funkcje do obsługi struktury listy

Do obsługi struktury listy zaimplementowano następujące funkcje:

- `int empty(list q)` - Sprawdza, czy lista jest pusta
- `void pop_front(list *h)` - Usuwa pierwszy element z listy
- `int return_front (list *h)` - Zwraca id pierwszego elementu z listy i go usuwa
- `void print_list(list h)` - Drukowanie listy
- `int size_list(list h)` - Funkcja zwracająca długość listy
- `void push(list *h, int a)` - Wstawianie wartości na początek listy
- `void remove_from_list(list *h, int u)` - Usuwanie elementu ze środka listy o podanym kluczu
- `int in_list (list h, int v)` - Sprawdza, czy element o podanym kluczu znajduje się w liście. tak==1, nie==0.
- `list copy_list (list h)` - Duplikowanie listy h
- `void reverse_list(list *h)` - Odwracanie listy. Jednoznaczne z przeczytaniem jej od tyłu

4.4 Implementacja DFS

4.4.1 Funkcja dfs_visit

Jest to funkcja odpowiadająca za akcję *odwiedzenia wierzchołka* - tj. pokolorowania go na szaro odkrywając go, zarejestrowania w nim czasy wejścia i wyjścia, kolorując wierzchołek na czarno po rekurencyjnym wykonaniu funkcji na sąsiadach do których da się dotrzeć z wierzchołka początkowego.

```
void dfs_visit(graph_p g, int u, int *time)
{
  // odkrywamy wierzchołek u
  g->info_v[u].color = gray;
  (*time)++;
  g->info_v[u].d = *time;

  // przeglądanie sąsiadów u wraz z ich potomkami
  list act_list;
  act_list = g->array[u].head;
  while (act_list != NULL){
    if (g->info_v[act_list->id].color == white){
      g->info_v[act_list->id].prev_node = u;
      dfs_visit(g, act_list->id, time);
    }
    act_list = act_list->next;
  }

  // opuszczamy wierzchołek u
  g->info_v[u].color = black;
  (*time)++;
  g->info_v[u].f = *time;
}
```

4.4.2 Algorytm DFS

Jest to przedłużenie funkcji `dfs_visit` na cały graf, upewniając się, że po zakończeniu wykonania każdy z wierzchołków G ma kolor czarny.

```
void dfs(graph_p g)
{
  // inicjalizacja
  int time = 0;
  for (int k=0; k < g->V; k++){
    g->info_v[k].color = white;
    g->info_v[k].prev_node = -1;
    g->info_v[k].d = -1;
    g->info_v[k].f = -1;
  }

  // budowa kolejnych drzew lasu przeszukiwania wgłąb
  for (int k = 0; k < g->V; k++)
    if (g->info_v[k].color == white)
      dfs_visit(g, k, &time);
}
```

4.5 Algorytm do szukania SSS

W celu ułatwienia szukania błędów w programie SSS zostały zaimplementowane pomocnicze wydruki pośrednich kroków, tj. wierzchołków wraz z właściwościami itp. Domyślnie są one wyłączone i uzależnione od globalnej zmiennej `tech`, lecz można je uruchomić ponownie poprzez przypisanie `tech = true`.

```
void sss(graph_p input)
{
  // Technikalia - deklarowanie zmiennych, inicjalizacje

  graph_p g = input;
  // Dorabiamy sobie graf transponowany g_T
  graph_p g_T = create_graph(g->V);
  transpose_graph(g, g_T);

  // inicjalizacja grafu g
  int time = 0;
  for (int k=0; k < g->V; k++){
      g->info_v[k].color = white;
      g->info_v[k].prev_node = -1;
      g->info_v[k].d = -1;
      g->info_v[k].f = -1;
  }

  // Na tę zmienną będziemy zapisywać transpozycję grafu G
  // inicjalizacja grafu g_T
  int time_T = 0;
  for (int k=0; k < g_T->V; k++){
      g_T->info_v[k].color = white;
      g_T->info_v[k].prev_node = -1;
```

```
    g_T->info_v[k].d = -1;
    g_T->info_v[k].f = -1;
}

/* Przeprowadzamy DFS na grafie G. Dostajemy z tego niezbędną
do dalszego działania listę czasów opuszczenia - f */
dfs(g);

// Techniczne wydruki kroku pośredniego
if (tech == true)
{
  printf("Po zrobieniu dfs na g\n\n");
  print_nodes(g);
  printf("\n");
}

  for(int i=0;i<g->V;i++)
{
  if (g->info_v[i].prev_node == -1)
  {
    if (tech == true) printf("Korzeń to: %d\n", i);
  }
}

// Listy czasów opuszczenia wierzchołka i odpowiadających im wierzchołków
list times;
list nodes;
ini(&times);
ini(&nodes);

int max_time = 0;
int max_time_node = 0;


/* Szukanie najpóźniejszego czasu opuszczenia wierzchołka - w celu
uzyskania warunku końcowego do pętli sortowania malejąco po czasie opuszczenia */
for(int i=0;i < g->V; i++)
{
  if (g->info_v[i].f > max_time)
  {
    max_time = g->info_v[i].f;
    max_time_node = i;
  }
}

push(&nodes, max_time_node);
push(&times, max_time);
if (tech == true)
  printf("Maksymalny czas to: %d dla wierzchołka %d\n", max_time, max_time_node);

// Sortowanie wierzchołków malejąco po czasie opuszczenia
int czas = 0;
```

```
int val = 0;
int tmp_max = 0;
int czas_big = max_time;
while (size_list(times) != g->V)
{
  for (int k=0; k < g->V;k++)
    {
      if (g->info_v[k].f < czas_big && g->info_v[k].f > tmp_max &&
      in_list(times, g->info_v[k].f) == 0)
      {
        czas = g->info_v[k].f;
        if (tech == true) printf("czas: %d\n", czas);
        val = k;
        tmp_max = czas;
      }
    else continue;
    }
  push(&times, tmp_max);
  push(&nodes, val);
  czas_big = tmp_max;
  tmp_max = 0;
}

if (tech == true)
{
  printf("\nCzasy:");
  print_list(times);
  printf("\nWęzły:");
  print_list(nodes);
}

// Odwracamy listę, by dostać prawidłowy porządek
reverse_list(&times);
reverse_list(&nodes);

if (tech == true)
{
  printf("\n\nPo odwróceniu\nCzasy:");
  print_list(times);
  printf("\nWęzły:");
  print_list(nodes);
  printf("\n");
}

/* Przeformułowanie algorytmu DFS, narzucając mu jako kolejność
przechodzenia kierunek malejących czasów, który wyznaczyliśmy wcześniej */
while(!empty(nodes))
{
  int k = return_front(&nodes);
  // Budowa kolejnych drzew lasu przeszukiwania wgłąb
  if (g_T->info_v[k].color == white)
      dfs_visit(g_T, k, &time_T);
```

```
  }

  if (tech == true)
  {
    printf("Po zrobieniu DFS na g_T w kolejności malejącego czasu f\n\n");
    print_nodes(g_T);
    printf("\n\n");
  }

  list ancestors;
  ini(&ancestors);
  for (int i=0; i<g->V;i++)
  {
    push(&ancestors, g->info_v[i].prev_node);
  }

  printf("Oto las silnie spójnych składowych grafu G:\n\n\n");

  if (tech == true)
  {
    printf("Przodkowie: ");
    print_list(ancestors);
    printf("\n");
  }
  int var;

  // Tutaj następuje wypisanie kolejnych drzew reprezentujących
  // szukane SSS - Silnie Spójnie Składowe
  for(int i=0;i<g->V;i++)
  {
    tree tmp;
    init_BST(&tmp);
    if (g_T->info_v[i].prev_node == -1)
    {
      var += 1;
      add_vertex_BST(&tmp, new_vertex(i));
      if (in_list(ancestors, i) == 0)
      {
        for (int j=0;j < g_T->V;j++)
        {
          if (g_T->info_v[j].d > g_T->info_v[i].d &&
          g_T->info_v[j].f < g_T->info_v[i].f) add_vertex_BST(&tmp, new_vertex(j));
        }
      }
      printf("Drzewo silnie spójnej składowej numer %d dla korzenia %d:\n", var, i);
        rysuj(tmp, 3);
      }
    }
}
```

5 Podsumowanie

Zapisany algorytm dla przykładowego grafu G przedstawionego na Rysunku 1 daje rezultat:

```
Oto las silnie spójnych składowych grafu G:


Drzewo silnie spójnej składowej numer 1 dla korzenia 0:
   0
Drzewo silnie spójnej składowej numer 2 dla korzenia 1:
         5
      2
   1
Drzewo silnie spójnej składowej numer 3 dla korzenia 3:
      6
   3
Drzewo silnie spójnej składowej numer 4 dla korzenia 4:
         8
      7
   4
Drzewo silnie spójnej składowej numer 5 dla korzenia 9:
   9
```

Wyniki te pokrywają się z oczekiwanymi, przedstawionymi na Rysunku 2. Oznacza to, że można z dużą dozą prawdopodobieństwa stwierdzić, iż udało się poprawnie zaimplementować algorytm przedstawiony nam na wykładzie. Na moment oddawania tego raportu nie znaleziono krytycznych błędów w działaniu programu, co oczywiście nie oznacza, że nie istnieją takie przykłady grafów dla których program zawiedzie. Będę wdzięczny za wszelkie sugestie jak mogę usprawnić program, lub poprawić jego funkcjonalności.

Spis treści

Literatura

[1] Algorytm Kosaraju

[2] Skrypt z wykładu Prof. Zawadowskiego