Podstawy Programowania 2

Grafy i ich reprezentacje

Arkadiusz Chrobot

Katedra Systemów Informatycznych

13 maja 2020

Plan

Wstęp

Grafy są w informatyce strukturami danych stosowanymi w wielu zagadnieniach. Zanim zaczęto je stosować w algorytmice, to były one wcześniej obecne w matematyce, gdzie pojęcie grafu wprowadził szwajcarski matematyk Leonard Euler. Stało się tak, kiedy rozwiązywał on problem siedmiu mostów w Królewcu. Grafy zapoczątkowały nową dziedzinę matematyki - topologię, a z czasem stały się częścią matematyki dyskretnej.
Zanim zajmiemy się reprezentacjami i zastosowaniami grafów poznamy kilka pojęć z nimi związanych. Niestety, nie istnieje jednolita terminologia dla tego zagadnienia, więc podane definicje w niektórych opracowaniach mogą być nieco inne.

Teoria grafów

Graf skierowany

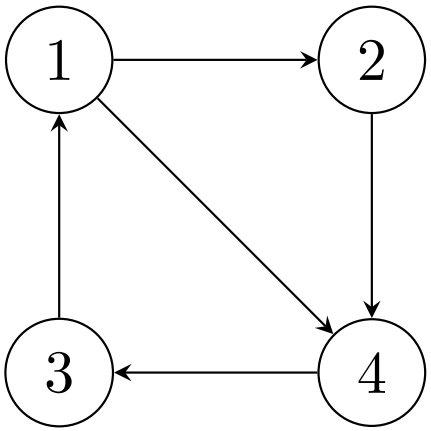
Graf skierowany G jest opisany parą (V, E), gdzie V jest zbiorem skończonym, którego elementy są wierzchołkami grafu G, a E jest relacją binarną w V i $E \subseteq V \times V$. Zbiór V nazywany jest krótko zbiorem wierzchołków, natomiast zbiór E nosi nazwę zbioru krawędzi G, a jego elementy nazywa się krawędziami.

Teoria grafów

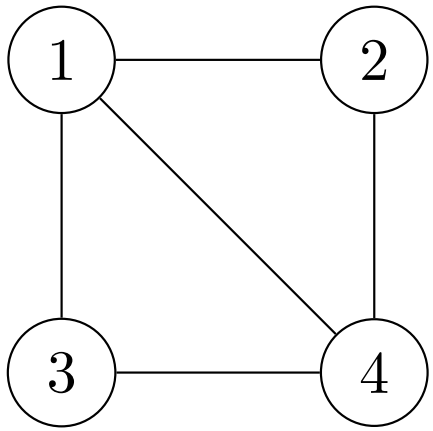
Graf nieskierowany

Graf nieskierowany jest grafem, którego zbiór E jest nieuporządkowany. Oznacza to, że krawędź jest zbiorem $\{u, v\}$, gdzie $u, v \in V$ i $u \neq v$. Krawędź oznaczamy używając zapisu (u, v). Zapisy (u, v) i (v, u) oznaczają tę samą krawędź. W grafie nieskierowanym nie mogą występować pętle, czyli krawędzie prowadzące do tego samego wierzchołka, z którego się zaczynają.

Teoria grafów



(a) Graf skierowany



(b) Graf nieskierowany

Przykłady grafów

Teoria grafów

Rodzaje krawędzi

W grafie skierowanym $G = (V, E)$ krawędź (u, v) jest krawędzią **wychodzącą** z wierzchołka u i **wchodzącą** do wierzchołka v. W grafie nieskierowanym taka krawędź (u, v) jest określana jako **incydentna** z wierzchołkami u i v.

Teoria grafów

Sąsiedztwo

Wierzchołek v jest **sąsiedni** do wierzchołka u w grafie $G = (V, E)$ jeśli łączy te wierzchołki krawędź (v, u). W grafie skierowanym *relacja sąsiedztwa* nie musi być symetryczna.

Teoria grafów

Stopień wierzchołka

Stopniem wierzchołka w grafie nieskierowanym jest liczba incydentnych z nim krawędzi. W grafie skierowanym **stopniem wejściowym** wierzchołka nazywamy liczbę krawędzi wchodzących do tego wierzchołka, a **stopniem wyjściowym** liczbę krawędzi z niego wychodzących. W grafie skierowanym **stopniem** wierzchołka jest suma stopnia wejściowego i wyjściowego.

Teoria grafów

Ścieżka

Ścieżka (droga) długości k z wierzchołka u do wierzchołka u' w grafie $G = (V, E)$ jest ciągiem wierzchołków $\langle v_0, v_1, v_2, \ldots, v_k \rangle$ takich, że $u = v_0$, $u' = v_k$ i $(v_{i-1}, v_i) \in E$ dla $i = 1, 2, \ldots, k$. Długość ścieżki jest liczbą krawędzi ścieżki. Ścieżka zawiera wierzchołki $v_0, v_1, v_2, \ldots, v_k$ i krawędzie $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$. Jeśli istnieje ścieżka z u do u', to mówimy, że u' jest osiągalny z u po ścieżce p. Ścieżka jest nazywana **ścieżką prostą** jeśli wszystkie jej wierzchołki są różne.

Teoria grafów

Cykle

Ścieżka $\langle v_0, v_1, v_2, \ldots, v_k \rangle$ tworzy **cykl** jeśli $v_0 = v_k$. Cykl nazywamy cyklem **prostym** jeśli dodatkowo wszystkie jego wierzchołki są różne. **Pętla** jest cyklem o długości 1. Graf skierowany nieposiadający pętli i krawędzi wielokrotnych (występujących więcej niż raz) nazywamy grafem **prostym**. Graf, który nie zawiera cykli nazywamy grafem **acyklicznym**.

Teoria grafów

Spójność

Graf nieskierowany jest **spójny** jeśli każda para jego wierzchołków jest połączona ścieżką. Graf skierowany **silnie spójny** to taki, w którym każde dwa wierzchołki są osiągalne jeden z drugiego.

Teoria grafów

Izomorfizm

Dwa grafy $G = (E, V)$ i $G' = (V', E')$ są **izomorficzne** jeśli istnieje wzajemnie jednoznaczne odwzorowanie $f : v \to v'$, takie że jeśli $(u, v) \in E$, to $(f(u), f(v)) \in E'$. Z tej własności grafów wynika, że mając graf nieskierowany możemy zastąpić go wersją skierowaną zamieniając każdą nieskierowaną krawędź na dwie przeciwnie skierowane krawędzie. Graf skierowany możemy zastąpić wersją nieskierowaną zamieniając każdą krawędź skierowaną krawędzią nieskierowaną i usuwając pętle.

Teoria grafów

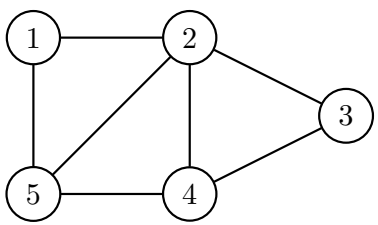
Grafy pełne i rzadkie

Graf *nieskierowany* nazywamy **grafem pełnym** jeśli każda para jego wierzchołków jest połączona krawędzią. Liczba krawędzi w takim grafie jest równa $\binom{n}{2}$, gdzie n jest liczbą wierzchołków grafu. Graf zawierający małą liczbę krawędzi w stosunku do liczby wierzchołków nazywamy **grafem rzadkim**.

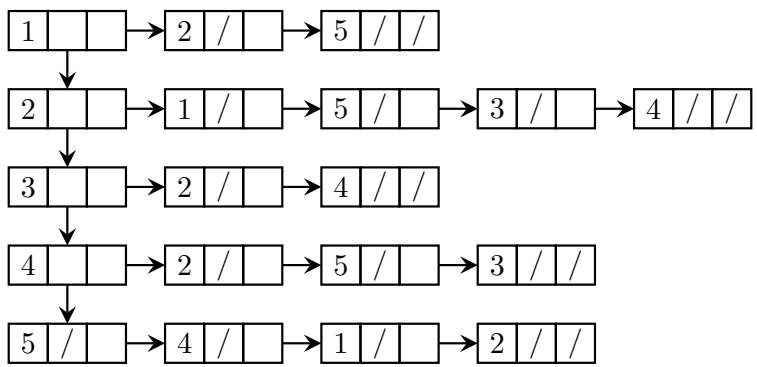
Grafy jako struktury danych

Istnieją dwa podstawowe sposoby reprezentowania grafów w pamięci komputera: za pomocą macierzy sąsiedztwa lub za pomocą listy sąsiedztwa. Lista sąsiedztwa może być zaimplementowana jako lista list lub jako tablica wskaźników na listy. Macierze sąsiedztwa to dwuwymiarowe tablice tworzone statycznie lub dynamiczne. Wiersze i kolumny w takiej macierzy reprezentują wierzchołki grafu. Jeśli dwa wierzchołki w grafie łączy krawędź, to w elemencie macierzy sąsiedztwa znajdującym się na przecięciu wiersza i kolumny odpowiadającym tym wierzchołkom zapisana jest liczba `1`, w przeciwnym razie umieszczona tam jest wartość `0`. Następne slajdy przedstawiają graf skierowany oraz nieskierowany i ich reprezentacje za pomocą listy sąsiedztwa oraz macierzy sąsiedztwa.

Reprezentacje grafu nieskierowanego



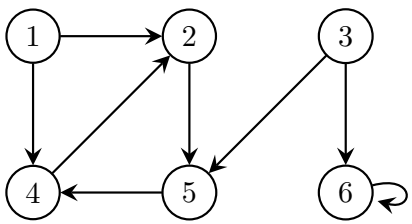
$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{bmatrix}$$



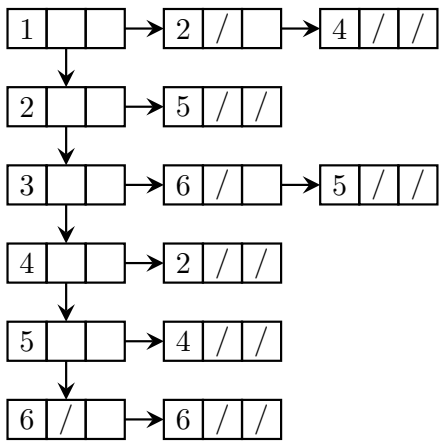
Reprezentacje grafu nieskierowanego

Po lewej stronie poprzedniego slajdu znajduje się ilustracja grafu nieskierowanego. W środku znajduje się macierz sąsiedztwa, a po prawej stronie lista sąsiedztwa zrealizowana jako lista list. Znaki / wewnątrz elementów listy oznaczają pola wskaźnikowe o wartości NULL. Proszę zwrócić uwagę, że macierz sąsiedztwa jest symetryczna względem swojej głównej przekątnej, a więc zachodzi równość $\mathbb{A} = \mathbb{A}^T$, gdzie $\mathbb{A}$ to macierz sąsiedztwa. Skoro ta macierz jest równa swojej macierzy transponowanej, to można zaoszczędzić miejsce w pamięci operacyjnej przechowując tylko jej elementy z macierzy trójkątnej górnej.

Reprezentacje grafu skierowanego



$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$



Reprezentacje grafu skierowanego

Podobnie jak w przypadku grafu nieskierowanego na poprzednim slajdzie przedstawiono kolejno (od lewej do prawej): ilustrację grafu, macierz sąsiedztwa i listę sąsiedztwa. Macierz sąsiedztwa jest nadal macierzą kwadratową, ale nie jest już symetryczna. Proszę także zwrócić uwagę, że graf posiada jedną krawędź, która jest pętlą. Jest ona w macierzy reprezentowana przez jedynkę znajdującą się na przecięciu szóstej kolumny i szóstego wiersza.

Reprezentacje grafów

Podsumowanie

Statystycznie rzecz ujmując częściej stosowaną reprezentacją grafów w informatyce jest lista sąsiedztwa. Jest ona implementowana jako lista list lub tablica list. Każdy element takiej tablicy lub listy (pionowa lista na ilustracjach z poprzednich slajdów), odpowiada jednemu wierzchołkowi grafu i wskazuje na listę wierzchołków, z którymi on sąsiaduje. Kolejność wierzchołków na tej ostatniej liście nie ma znaczenia. Suma długości wszystkich list sąsiedztwa wynosi w przypadku grafu skierowanego $|E|$, a w przypadku grafu nieskierowanego $2 \cdot |E|$, gdzie zapis $|E|$ oznacza liczebność zbioru krawędzi grafu. Reprezentacja za pomocą listy sąsiedztwa wymaga zatem $O(V + E)$ pamięci, natomiast macierz sąsiedztwa wymaga $\Theta(V^2)$. Obie reprezentacje mogą być używane do reprezentowania zarówno grafów z wagami, jak i grafów bez wag. W tym ostatnim przypadku można zaoszczędzić pamięć potrzebną na macierz sąsiedztwa zapisując wartość każdego jej elementu na pojedynczym bicie. Jest to oszczędność pamięci kosztem czasu wykonania.

Reprezentacje grafów

Podsumowanie

Macierze sąsiedztwa lepiej się sprawdzają od list w zagadnieniach polegających na ustalaniu, czy istnieje krawędź między wierzchołkami grafu lub (pod warunkiem, że liczba wierzchołków jest niezmienna) na dodawaniu nowych krawędzi do grafu albo usuwaniu istniejących. Z kolei listy sąsiedztwa są bardziej przydatne w zagadnieniach związanych z przechodzeniem grafu (większość algorytmów grafowych wykonuje tę czynność) lub znajdywaniem stopnia wierzchołków. Również lepiej nadają się one do reprezentowania grafów małych lub rzadkich. Macierze sąsiedztwa są nieco lepszym rozwiązaniem, jeśli chcemy reprezentować w pamięci komputera gęste grafy.

Obie reprezentacje są wymienne, tzn. macierz sąsiedztwa może zostać zastąpiona listą sąsiedztwa i odwrotnie. Na kolejnych slajdach przedstawiony jest program, który konwertuje zaprezentowaną wcześniej macierz sąsiedztwa grafu nieskierowanego na listę sąsiedztwa.

Grafy jako struktury danych

Macierz sąsiedztwa i typ bazowy listy sąsiedztwa

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  typedef int matrix[5][5];
5
6  const matrix adjacency_matrix = {{0,1,0,0,1},
7                                   {1,0,1,1,1},
8                                   {0,1,0,1,0},
9                                   {0,1,1,0,1},
10                                  {1,1,0,1,0}};
11
12 struct vertex
13 {
14     int vertex_number;
15     struct vertex *next, *down;
16 } *start_vertex;
```

Grafy jako struktury danych

Macierz sąsiedztwa i typ bazowy listy sąsiedztwa

W programie będą użyte funkcje z plików nagłówkowych `stdio.h` oraz `stdlib.h`. W wierszu nr 4 zdefiniowany jest typ tablicowy dla macierzy (tablica dwuwymiarowa, kwadratowa o 25 elementach). W wierszach 6-10 utworzona jest macierz sąsiedztwa dla grafu nieskierowanego bez wag. W wierszach 12-16 zdefiniowano typ bazowy listy sąsiedztwa (listy list). Pole wskaźnikowe `down` będzie używane do łączenia elementów w listę wszystkich wierzchołków („lista pionowa"), a pole `next` do łączenia elementów na listach sąsiedztwa („listy poziome"). Dodatkowo w wierszu nr 16 zadeklarowana jest zmienna wskaźnikowa, która będzie wskazywała na element reprezentujący wierzchołek startowy (`start_vertex`) [1]. Jest to zmienna globalna, zatem jej wartość początkowa wynosi `NULL`.

[1] Na ilustracji jest to element w lewym górnym rogu.

Grafy jako struktury danych

Funkcja `create_vertical_list()`

```
1  void create_vertical_list(struct vertex **start_vertex,
2                            const matrix adjacency_matrix)
3  {
4      int i;
5      for(i=0; i<sizeof(matrix)/sizeof(*adjacency_matrix); i++) {
6          *start_vertex = (struct vertex *)
7                          malloc(sizeof(struct vertex));
8          if(*start_vertex) {
9              (*start_vertex)->vertex_number = i+1;
10             (*start_vertex)->down = (*start_vertex)->next = NULL;
11             start_vertex = &(*start_vertex)->down;
12         }
13     }
14 }
```

Grafy jako struktury danych

Funkcja `create_vertical_list()`

Funkcja `create_vertical_list()` tworzy listę „pionową”, czyli listę wszystkich wierzchołków grafu. Nie zwraca ona żadnej wartości. Przez jej pierwszy parametr, który jest podwójnym wskaźnikiem, będzie przekazany jako pierwszy argument jej wywołania adres zmiennej `start_vertex`. Ten parametr będzie również wykorzystywany wewnątrz tej funkcji w innych celach. Przez drugi parametr tej funkcji przekazywana jest macierz sąsiedztwa. Jest to przekazanie przez stałą, gdyż zawartość tej macierzy nie ulega zmianie. Lista „pionowa” jest tworzona wewnątrz pętli `for`. Liczba iteracji tej pętli jest określona przez liczbę wierzchołków grafu, czyli przez liczbę wierszy macierzy sąsiedztwa, która jest określana za pomocą wyrażenia, w którym rozmiar typu jest dzielony przez rozmiar wiersza (dereferencja wskaźnika na tablicę dwuwymiarową daje dostęp do pojedynczego wiersza tej tablicy).

Grafy jako struktury danych

Funkcja `create_vertical_list()`

W pętli `for` tej funkcji przydzielana jest pamięć na kolejne elementy listy „pionowej”. Jeśli przydział się kończy sukcesem, to do pola `vertex_number` elementu wpisywany jest numer wierzchołka (wartość licznika pętli powiększona o jeden, gdyż numeracja wierzchołków zaczyna się od jedynki, a numeracja elementów macierzy do zera) i inicjowane są oba pola wskaźnikowe (wiersz nr 10), a następnie do wskaźnika `start_vertex` zapisywany jest adres pola wskaźnikowego `down` tego elementu (wierszy nr 11). Dzięki temu rozwiązaniu nie trzeba osobno oprogramowywać przypadku, gdy tworzony jest pierwszy element listy. Po zakończeniu pętli ten wskaźnik wskazuje na pole `down` ostatniego elementu na liście.

Grafy jako struktury danych

Funkcja `convert_matrix_to_list()`

```
1   struct vertex *convert_matrix_to_list(const matrix adjacency_matrix)
2   {
3       struct vertex *start_vertex = NULL;
4       create_vertical_list(&start_vertex,adjacency_matrix);
5       if(start_vertex) {
6           struct vertex *horizontal_pointer = NULL, *vertical_pointer = NULL;
7           horizontal_pointer = vertical_pointer = start_vertex;
8           int i,j;
9           for(i=0; i<sizeof(matrix)/sizeof(*adjacency_matrix); i++) {
10              for(j=0; j<sizeof(matrix)/sizeof(*adjacency_matrix); j++)
11                  if(adjacency_matrix[i][j]) {
12                          struct vertex *new_vertex = (struct vertex *)malloc(sizeof(struct vertex));
13                          if(new_vertex) {
14                              new_vertex->vertex_number = j+1;
15                              new_vertex->down = new_vertex->next = NULL;
16                              horizontal_pointer->next = new_vertex;
17                              horizontal_pointer = horizontal_pointer->next;
18                          }
19                  }
20              vertical_pointer = vertical_pointer->down;
21              horizontal_pointer = vertical_pointer;
22          }
23      }
24      return start_vertex;
25  }
```

Grafy jako struktury danych

Funkcja `convert_matrix_to_list()`

Funkcja `convert_matrix_to_list()` dokonuje właściwej konwersji macierzy sąsiedztwa do listy. Jako wynik swojego działania zwraca ona wskaźnik do elementu listy list reprezentującego wierzchołek startowy, a jako argument wywołania przyjmuje przez swój jedyny parametr macierz sąsiedztwa. Jest to przekazanie przez stałą. Funkcja ta posiada lokalną zmienną wskaźnikową `start_vertex`, która jest inicjowana wartością `NULL`. Tworzy ona „listę pionową” wywołując funkcję `create_vertical_list()` (wiesz nr 4). Jeśli utworzenie tej listy się powiedzie, co jest sprawdzane w wierszu nr 5, to funkcja rozpoczyna iterację po wszystkich elementach macierzy w dwóch pętlach `for`. Wcześniej zadeklarowane i zainicjowane są wskaźniki `horizontal_pointer` i `vertical_pointer` (wiersze nr 6 i 7). Pierwszy będzie wykorzystywany do iterowania po „listach poziomych”, a drugi po „liście pionowej”.

Grafy jako struktury danych

Funkcja `convert_matrix_to_list()`

Zewnętrzna pętla `for` iteruje po wierszach macierzy sąsiedztwa, a wewnętrzna po kolumnach. Wartość indeksu kolumny powiększona o jeden oznacza numer wierzchołka grafu, z jakim potencjalnie sąsiaduje wierzchołek określany przez wartość indeksu wiersza. Wewnątrz zagnieżdżonej pętli `for` funkcja sprawdza, czy wartość bieżącego elementu macierzy jest różna od zera (wiersz nr 11). Jeśli tak, to tworzony jest nowy element listy list, który będzie reprezentował wierzchołek sąsiadujący (wiersz nr 12). Jeśli jego utworzenie się powiedzie, to do tego elementu zapisywany jest numer tego wierzchołka (wiersz nr 14) i inicjowane są jego pola wskaźnikowe (wiersz nr 15), a następnie ten element dodawany jest do listy sąsiedztwa (listy „poziomej") bieżącego wierzchołka grafu (wiersze 16 i 17). W tej ostatniej czynności wykorzystywany jest wskaźnik `horizontal_pointer`, który wskazuje na ostatni (początkowo również pierwszy) element listy sąsiedztwa.

Grafy jako struktury danych

Funkcja `convert_matrix_to_list()`

Po zakończeniu wszystkich iteracji wewnętrznej pętli `for` do wskaźnika `vertical_pointer` zapisywany jest adres kolejnego elementu na „pionowej liście”, czyli liście wierzchołków grafu (wiersz nr 20). Adres tego samego elementu jest również zapisywany we wskaźniku `horizontal_pointer`. Po zakończeniu obu pętli zwracany jest przez funkcję adres wierzchołka startowego, po czym kończy ona swoje działanie.

Grafy jako struktury danych

Funkcja `print_adjacency_list()`

```
1  void print_adjacency_list(struct vertex *start_vertex)
2  {
3      while(start_vertex) {
4          printf("%3d:",start_vertex->vertex_number);
5          struct vertex *horizontal_pointer = start_vertex->next;
6          while(horizontal_pointer) {
7              printf("%3d",horizontal_pointer->vertex_number);
8              horizontal_pointer = horizontal_pointer->next;
9          }
10         start_vertex = start_vertex->down;
11         puts("");
12     }
13 }
```

Grafy jako struktury danych

Funkcja `print_adjacency_list()`

Zadaniem funkcji, której kod źródłowy znajduje się na poprzednim slajdzie jest wypisanie zawartości listy sąsiedztwa, w takim porządku jak na ilustracjach znajdujących się na wcześniejszych slajdach. Funkcja ta nic nie zwraca, ale przyjmuje przez parametr wskaźnik na element tej listy reprezentujący wierzchołek startowy. W jej treści umieszczone są dwie pętle `while`. Zewnętrzna iteruje po liście wierzchołków („liście pionowej”), a wewnętrzna po listach sąsiedztwa związanych z danymi wierzchołkami (o ile nie są one puste). Warunkiem wykonania pętli zewnętrznej (wiersz nr 3) jest to, że wskaźnik `start_vertex` nie ma wartości `NULL`. Jeśli jest on spełniony, to wypisywany jest numer wierzchołka umieszczony w elemencie listy wierzchołków (wiersz nr 4), a następnie inicjowany jest zadeklarowany w wierszu nr 5 wskaźnik na listę sąsiedztwa. Jeśli jego wartość również jest różna od `NULL`, to rozpoczyna się działanie wewnętrznej pętli `while` (wiersz nr 6).

Grafy jako struktury danych

Funkcja `print_adjacency_list()`

W tej pętli wypisywane są numery wierzchołków umieszczone na liście wierzchołków sąsiadujących (wiersz nr 7). Do poruszania się po niej wykorzystywany jest wskaźnik `horizontal_pointer`, któremu w kolejnych iteracjach pętli wewnętrznej przypisywany jest adres kolejnych elementów tej listy (wiersz nr 8). Tuż po zakończeni pętli wewnętrznej w pętli zewnętrznej wskaźnikowi `start_vertex` przypisywany jest adres kolejnego elementu na „liście pionowej” (wiersz nr 10) i kursor przemieszczany jest do kolejnego wiersza na ekranie (wiersz nr 11).

Grafy jako struktury danych

Funkcja `remove_adjacency_list()`

```
1  void remove_adjacency_list(struct vertex **start_vertex)
2  {
3      while(*start_vertex) {
4          struct vertex *horizontal_pointer = (*start_vertex)->next;
5          while(horizontal_pointer) {
6              struct vertex *next_horizontal =
7                                              horizontal_pointer->next;
8              free(horizontal_pointer);
9              horizontal_pointer = next_horizontal;
10         }
11         struct vertex *next_vertical = (*start_vertex)->down;
12         free(*start_vertex);
13         *start_vertex= next_vertical;
14     }
15 }
```

Grafy jako struktury danych

Funkcja `remove_adjacency_list()`

Funkcja `remove_adjacency_list()`, która usuwa listę sąsiedztwa z pamięci komputera ma podobną konstrukcję do funkcji opisywanej poprzednio. Podobnie jak `print_adjacency_list()` nie zwraca ona żadnej wartości, ale ma parametr w postaci podwójnego wskaźnika na listę sąsiedztwa. Wewnątrz funkcji również wykonywane są dwie pętle `while`. W pierwszej, jeśli lista sąsiedztwa jest niepusta (wiersz nr 3) to inicjowany jest zadeklarowany w wierszu nr 4 wskaźnik `horizontal_pointer`. Jeśli jego wartość będzie różna od `NULL`, to wykonywana jest wewnętrzna pętla `while`, w której zwalniana jest lista wierzchołków sąsiadujących z wierzchołkiem, który jest reprezentowany przez element bieżąco wskazywany przez wskaźnik `start_vertex`.

Grafy jako struktury danych

Funkcja `remove_adjacency_list()`

Zwalnianie to odbywa się zgodnie z algorytmem znanym z listy jednokierunkowej, tzn. najpierw zapamiętywany jest we wskaźniku `next_horizontal` adres kolejnego elementu na liście wierzchołków sąsiednich (wiersze nr 6 i 7), następnie zwalniany jest element wskazywany przez `horizontal_pointer` (wiersz nr 8) i temu wskaźnikowi przypisywany jest adres z `next_horizontal` (wiersz nr 9). Po zwolnieniu całej listy wierzchołków sąsiadujących w pętli zewnętrznej usuwany jest element z listy wszystkich wierzchołków („listy pionowej") reprezentujący wierzchołek, z którym te z listy „poziomej" sąsiadowały. Sposób zwalniania elementów z listy wszystkich wierzchołków jest taki sam, jak dla listy wierzchołków sąsiadujących. Najpierw zapamiętywany jest adres następnego elementu na liście w zmiennej `next_vertical` (wiersz nr 11). Następnie zwalniana jest pamięć elementu wskazywanego przez `start_vertex` (wiersz nr 12) i temu wskaźnikowi jest przypisywany adres zapamiętany w `next_vertical` (wiersz nr 13).

Grafy jako struktury danych

Funkcja `remove_adjacency_list()`

Po zakończeniu obu pętli `while` lista sąsiedztwa jest całkowicie usunięta z pamięci operacyjnej komputera. A wskaźnik na nią ma wartość `NULL`.

Grafy jako struktury danych

Funkcja `main()`

```
1  int main(void)
2  {
3      start_vertex = convert_matrix_to_list(adjacency_matrix);
4      if(start_vertex) {
5          print_adjacency_list(start_vertex);
6          remove_adjacency_list(&start_vertex);
7      }
8      return 0;
9  }
```

Grafy jako struktury danych

Funkcja `main()`

Na początku funkcji `main()` programu, na bazie macierzy sąsiedztwa tworzona jest lista sąsiedztwa, przy pomocy wywołania funkcji `convert_matrix_to_list()` (wiersz nr 3). Jeśli ta lista nie jest pusta, co sprawdzane jest w wierszu nr 4, to najpierw wypisywana jest jej zawartość przy pomocy `print_adjacency_list()`, a następnie jest ona usuwana z pamięci komputera przez wywołanie funkcji `remove_adjacency_list()`. Po tej czynności funkcja zwraca wartość `0` i kończy swoje działanie. Tym samym kończy się działanie całego programu.

Zastosowania grafów

Grafy są prostym formalizmem, który może być stosowany do znajdywania rozwiązań wielu problemów. Najczęściej są to zagadnienia, w których występuje potrzeba odwzorowania relacji. Przykładem może być analiza sieci społecznych (ang. *social networks*). Poza tym grafy stosowane są do modelowania układów elektronicznych, topologi układów scalonych VLSI, systemów dróg zarówno lądowych, jak i wodnych oraz powietrznych, a także sieci telekomunikacyjnych. Niepodważalną zaletą stosowania grafów w informatyce jest to, że istnieje wiele gotowych i efektywnych algorytmów powiązanych z tymi strukturami danych. Więcej na temat tego zagadnienia można przeczytać w książce „Wprowadzenie do algorytmów" autorstwa T. H. Cormena, Ch.E. Leisersona i R. Rivesta oraz w "The Algorithm Design Manual" Stevena S. Skieny.

Pytania

?

KONIEC

Dziękuję Państwu za uwagę!