Mateusz Krawczyk

Zad.12 - Stacje BTS - dokumentacja

Treść zadania.

Zadanie polega na pokryciu wejściowej planszy, która jest postaci np.

```
* * * o o
o o * * *
```

, gdzie * oznaczają wieże do których doprowadzony ma być sygnał. Należy pokryć wszystkie * za pomocą "prostokątów" 1x2. Pokrywać można tylko pionowo lub poziomo i należy wykorzystać jak najmniejszą liczbę pokryć, np. dla powyższego przykładu będą to minimum 3 pokrycia.

Opis rozwiązania.

Zadanie można przedstawić za pomocą grafów. Każda * w wejściowych danych to wierzchołek w grafie. Wtedy każde możliwe pokrycie dwóch sąsiednich * będzie po prostu krawędzią poprowadzoną między tymi dwoma wierzchołkami w tworzonym grafie. Jeśli * nie ma sąsiadów, natychmiast można dodać +1 pokrycie do wyniku i ją usunąć. Po stworzeniu grafu zadanie sprowadza się do znalezienia tzw. **minimum edge cover**, czyli pokrycia najmniejszej liczby krawędzi w taki sposób, aby każdy wierzchołek był początkiem lub końcem, co najmniej jednej pokrytej krawędzi.
Z drugiej strony można rozwiązać zadanie od strony wierzchołków przy pomocy tzw. **vertex cover**, czyli pokrycia wierzchołkowego w taki sposób, by każda krawędź w grafie była połączona z co najmniej jednym pokrytym wierzchołkiem. Interesujące będzie tu pokrycie przy pomocy najmniejszej liczby wierzchołków - **minimum vertex cover**. Nie będzie to jednak rozwiązaniem zadania. Do rozwiązania zadania będzie potrzebne tzw. **minimal vertex cover**. Jest to pokrycie wierzchołków w taki sposób, że usunięcie dowolnego wierzchołka z tego zbioru sprawi, że nie będzie to już pokrycie wierzchołkowe(**vertex cover**).

Dla przykładu graf składający się z trzech wierzchołków i dwóch krawędzi.

1 - 2 - 3

Dla powyższego grafu **minimum vertex cover** wynosi 1 i jest to pokrycie wierzchołka **2**, zaś **minimal vertex cover** jest równe 2 dla wierzchołków **1** i **3**, gdyż usunięcie np. **1** sprawi, iż krawędź (**1**-**2**) nie będzie połączona z żadnym pokrytym wierzchołkiem, a więc nie będzie to **vertex cover** dla tego grafu.

Aby uzyskać **minimal vertex cover** wystarczy odjąć od ilości wierzchołków w grafie **V**, wartość **minimum vertex cover** tego grafu. Jest ono równoważne **minimum edge cover**, a więc stanowi rozwiązanie zadania. Zadanie sprowadza się tym samym do problemu znalezienia **minimum vertex cover.** Dla standardowych grafów jest to zadanie o złożoności NP – complete. Są jednak grafy w których można znaleźć to rozwiązanie w czasie wielomianowym. Są to drzewa oraz grafy dwudzielne. Można łatwo zauważyć, że dla pokrycia o długości 2, stworzony graf będzie grafem dwudzielnym – wszystkie * można oznaczyć jako 1 lub 2 – stanowiące grupy dwudzielności, w taki sposób, że żadne dwie 1-ki i 2-ki nie sąsiadują ze sobą. Istnieją znane algorytmy rozwiązujące problem dla podanych grafów.

W rozwiązaniu wykorzystany został zaimplementowany algorytm Hopcroft-Karp, który opiera się na własności maximum matching w grafie. Matching(skojarzenie) to podzbiór krawędzi M, w którym co najwyżej jedna krawędź jest incydentna z każdym wierzchołkiem V. Jeśli wierzchołek v jest incydentny do pewnej krawedzi M, to jest skojarzony, w przeciwnym wypadku jest wolny. Podobnie gdy krawęź e należy do matching to jest skojarzona, w przeciwnym wypadku jest wolna. Maximum matching to taki podzbiór krawędzi, w którym zawarta jest maksymalna liczba takich skojarzeń. Algorytm Hopcorft-Karp znajduje maximum matching posługując się augmenting paths(ścieżkami powiększającymi). Jest to taka ścieżka p, ze jej krawędzie są na przemian skojarzone oraz wolne. Łatwo zauważyć, że jeżeli istnieje augmenting path względem M, to M nie jest maximum matching. Mozna skonstruować wtedy większe matching, zamieniajac na ścieżce krawędzie wolne na skojarzone i na odwrót. To, że matching M jest maksymalne, gdy nie istnieje względem niego żadna ścieżka powiększająca zostało udowodnione w twierdzeniu Berge’a. Normalne wyszukiwanie jednej ścieżki ma złożoność O(V*E). W algorytmie Hopcroft-Karp metoda korzystająca z augmenting paths została przyśpieszona, bo zamiast

wyszukiwać ścieżki pojedynczo, wiele ścieżek jest szukanych naraz. Za pomocą dowodów można wykazać, że złożoność algorytmu wynosi **O(√V*E)**.
Ogólna zasada algorytmu polega na znajdywaniu augmenting paths i dodawaniu znalezionych ścieżek do aktualnego matching, czyli zamianie krawędzi, które były skojarzone na wolne oraz odrwotnie – wolnych na skojarzone.
Do znalezienia augmenting paths wykorzystany zostaje algorytm BFS(Breadth First Search). Po znalezieniu takiej ścieżki wykorzystany zostaje algorytm DFS(Depth First Search), by dodać ją do aktualnego matching.
W algorytmie wykorzystane zostały dynamicznie alokowane listy oraz kolejka queue. W całym programie głównie wykorzystywane są dynamicznie alokowane tablice dwuwymiarowe.

Wykonanie programu.

Na wejściu dostajemy tablicę 2-wymiarową T wypełnioną * oraz o. Tworzone są dwie pomocnicze, dynamicznie alokowane tablice dwuwymiarowe T1 oraz T2.
T1 dla odpowiadających komórek w T, zawiera liczby:
0 – gdy w T jest to o
1 – gdy w T jest to *, wtedy ta * jest wierzchołkiem należącym do pierwszej z dwóch grup grafu dwudzielnego
2 – jak dla 1, ale * należy do drugiej grupy
3 – jeśli dana * nie sąsiaduje z innymi *, nie zostaje ona uwzględniona w grafie

T2 zawiera odpowiednią numerację dla wierzchołków które w T1 mają odpowiednio nadaną wartość 1(od 1 do v1) oraz analogicznie dla 2(1 do v2). Ogólne V = v1+v2;

Poszczególne funkcje potrzebne do rozwiązania zadania są zdefiniowane w cover.cpp, a uruchamiane w poszczegolnych trybach w main.cpp.
W cover.cpp w funkcji **fillArrays()** T1 wypełnione jest na przemian 1 i 2, tak by te same ze sobą nie sąsiadowały. W T2 wszystkie pola mają wartość 0.
Graf tworzony jest w funkcji makeGraph**(string**T)**, która na wejściu dostaje tablicę T. Przy pomocy T1 oraz T2 tworzona jest numeracja oraz podział przyszłych wierzchołków. * oznaczone jako 3 w T1, czyli nie posiadające sąsiadów nie są uwzględniane w grafie. Po prostu mają pokrycie jednostkowe, a więc ogólne rozwiązanie Cover jest inkrementowane dla każdego z nich.
Następnie stworzony zostaje sam graf jako obiekt klasy **Graph(v1, v2)**,

gdzie v1 oraz v2 oznaczają liczbę wierzchołków w dwóch grupach dwudzielności, co wykorzystane zostało przy ich numeracji od 1 do v1 oraz od 1 do v2. Krawędzie grafu tworzone za pomocą metody **Graph::addEdge(int u, int v)**, gdzie u i v oznaczają numery odpowiednich wierzchołków z obu grup dwudzielności połączonych krawędzią. Funkcja zwraca wskaźnik do stworzonego grafu.
Następnie wywoływany zostaje sam algorytm funkcją algorithm(Graph *g) dostający jako argument stworzony wcześniej graf. Szuka on maximum matching – Max, dla otrzymanego grafu. Następnie obliczone **V – Max**, które zwróci wartość minimal edge cover dodawane jest do aktualnego Cover, które jest liczbą równą ilości wierzchołków, które nie miały sąsiada i mają pokrycie jednostkowe lub po prostu równe 0. Rozwiązaniem zadania jest ostatecznie wartość **Cover.**

Opis kodu.

Main

main.cpp - główny plik źródłowy

Zadeklarowane następujące zmienne globalne:
- *int N* – liczba wierszy w grafie
- *int M* – liczba kolumn w grafie
- *Timer timer* – obiekt timer do mierzenia czasu
- *Graph *g* – wskaźnik na stworzony graf

Funkcje:
- *double funcT(int vv)* – zwraca szacunkową złożoność algorytmu

Cover

cover.cpp – plik źródłowy zawierający funkcje wykonujące zadanie
cover.h – plik nagłówkowy

Zadeklarowane następujące zmienne globalne:
- *int Cover* – wynikowe pokrycie
- *int V* – liczba wierzchołków w grafie, E – liczba krawędzi w grafie
- *int Max* – wynik algorytmu Hopcrofta-Karpa do znalezienia maximum matching
- *int **T1, **T2* – wskaźniki do pomocniczych tablic dla tworzenia grafu

Funkcje:
- *double funcT(int vv)* – zwraca szacunkową złożoność algorytmu
- *void fillArrays()* - tworzy pomocnicze tablice T1 oraz T2
- *Graph* makeGraph(string** T)* – funkcja tworząca graf dwudzielny, jako argument dostaje podstawową Tablicę T wy:pełnioną * oraz o. Modyfikuje wynik Cover dodając * bez sąsiadów
- *int algorithm(Graph *g)* – funkcja wykonująca algorytm Hopcrofta-Karpa dla podanego jako argument grafu, zwraca ostateczny wynik pokrycia
W pliku nagłówkowym zdefiniowane dwie pomocnicze funkcje:
- *template<typename T>*
*void show(T** tab) / void clear(T** tab)* – funkcje służące do pokazania/ wyczyszczenia tablicy dwu wymiarowej typu T(string/int)

Graph
graph.cpp – plik źródłowy grafu
graph.h – plik nagłówkowy grafu

Zawiera klasę Graph. Jej zmienne prywatne:
- *int m, n* – liczba wierzchołków dla lewego i prawego podzbioru klas dwudzielności
- *list<int> *adj* – 'lista list', przechowuje graf jako odpowiednie wierzchołki połączone krawędzią
- *int *left, *right, *dist* – wskaźniki dla tablic, wykorzystanych do algorytmu Hopcrofta-Karpa

Metody:
- *Graph(int m, int n)* – konstruktor grafu
- *void addEdge(int u, int v)* – dodaje krawędź między wierzchołkami u i v
- *bool bfs()* - algorytm BFS wykorzystany do znajdywania augmenting paths
- *bool dfs(int u)* – algorytm DFS, przechodzi po ścieżce wyznaczonej przez BFS jeśli zaczyna się ona w wierzchołku u i odpowiednio dodaje ją do matching
- *int matching()* - główny algorytm Hopcrofta-Karpa
- *void showGraph()* - metoda pomocnicza do wyświetlenia grafu

Generator
generator.cpp – plik źródłowy generatora
generator.h – plik nagłówkowy generatora

Zawiera funkcje:
- *string **generate(int n, int m, int k, int l)* – generuje ona losową tablicę wypełnioną * oraz o, wielkości n*m, posiadającą pokrycie minimalne k. l – oznacza ile * można pokryć naraz pionowo/poziomo – w zadaniu l=2

Timer
Timer.h – plik nagłówkowy dla timera

Zawiera klasę Timer. Służy ona do mierzenia czasu wykonania algorytmu, korzystając z std::chrono.

Zmienne:
- *Clock::time_point epoch* – ustalony czas

Metody:
- *void start()* - rozpoczyna odliczanie, zapisuje moment w czasie w zmiennej epoch
- *Clock::duration time_elapsed() const* – zwraca różnicę czasu aktualnego oraz zapisanego przez start() w epoch

Złożoność programu.

Wykonanie się programu, czyli wyliczenie pokrycia, bez uwzględnienia wygenerowania danych w generatorze, zawsze wywoła następujące funkcje:

- *fillArrays()*, które ustawia pomocnicze tablice dwuwymiarowe T1 oraz T2 – złożoność rzędu **M*N+c**
- *makeGraph(string **T)* – stworzenie samego grafu ma złożoność **2*M*N + c**, gdzie c to jakaś stała, gdyż dwa razy przechodzi przez całe tablice T1 oraz T2, c to stała potrzebna na instrukcje podstawowe, głównie porównania i przypisania.
- *algorithm(Graph *g)* – sam algorytm liczący pokrycie. Wykorzystany w nim algorytm Hopcrofta-Karpa zapewnia złożoność **√V*E.**

Ogólnie złożoność całości wynosi więc **T(N,M,E,V) = 3*M*N+c+√V*E.**
V oraz E są losowe, zależne od wygenerowanej tablicy. V nie ma w zasadzie ograniczenia dolnego, może wynosić nawet 0, jeśli żadna * w wygenerowanej tablicy nie ma sąsiada. Jest jednak ograniczone z góry przez M*N, bo tyle maksymalnie może być wygenerowanych *, czyli wierzchołków. Ostatni człon

może się więc wyzerować albo być bardzo mały w wielu przypadkach. Za każdym razem wykonywana jest jednak wypełnianie fillArrays() - N*M oraz początkowe sprawdzenie grafu w celu wytworzenia go w funkcji makeGraph(), które wykonuje się w pętli N*M. Złożoność jest więc zawsze ograniczona od dołu przez **2*M*N**, a więc złożoność całego programu jest rzędu **O(M*N).**
W najgorszym przypadku, czyli gdy cała tablica jest wypełniona, a **V=M*N**, można założyć, że **E<4*M*N**, ponieważ * w rogach mogą mieć maksymalnie 2 sąsiadów, pozostałe * na krawędziach po 2-3, a te w środku <=4, w zależności od sposobu liczenia, czyli **√V*E** będzie rzędu **√(M*N)*M*N = O((M*N)^(1.5)) –** złożoność pesymistyczna programu.

Tabelki wygenerowana w 3-im trybie wykonania z pomiarem czasu i złożoności dla **T(n,m) = n*m**

Wywołanie opisane w readme.txt.
Dla danych startowych N=M=500, Cover = 100, k = 20, step = 100, r = 10

n	m	t(n,m)[ms]	q(n,m)
500	500	6.40077	1.05049
600	600	8.78441	1.00117
700	700	12.1045	1.01356
800	800	16.3538	1.04843
900	900	20.5602	1.04146
1000	1000	29.1177	1.1947
1100	1100	36.1961	1.22737
1200	1200	41.5629	1.18425
1300	1300	42.7125	1.03697
1400	1400	49.0539	1.02687
1500	1500	54.8381	1
1600	1600	62.204	0.996961
1700	1700	69.9629	0.993275
1800	1800	78.2341	0.990719
1900	1900	85.9724	0.977128
2000	2000	95.0444	0.974914
2100	2100	104.907	0.97604
2200	2200	117.119	0.992847
2300	2300	128.231	0.994573
2400	2400	139.316	0.992384

Dla różnych M = 300 oraz N = 700 i większego Cover = 1000

./cover -m3 300 700 1000 20 100 10

n	m	t(n,m)[ms]	q(n,m)
300	700	5.64169	1.14049
400	800	8.15744	1.0822
500	900	11.3946	1.07495
600	1000	14.5794	1.03155
700	1100	18.5274	1.02147
800	1200	22.9579	1.01523
900	1300	27.987	1.01548
1000	1400	33.3386	1.01093
1100	1500	39.0918	1.00578
1200	1600	45.4957	1.00594
1300	1700	52.0584	1
1400	1800	58.9092	0.992395
1500	1900	66.7101	0.993684
1600	2000	75.4292	1.00067
1700	2100	84.3093	1.00256
1800	2200	92.4846	0.991462
1900	2300	102.819	0.998831
2000	2400	112.499	0.994969
2100	2500	121.977	0.986326
2200	2600	134.472	0.998015