Algorytmy i struktury danych

Laboratorium 5

Termin wysłania (MS Teams): **07 czerwca 2021 godz. 13:14**

Tematy, których dotyczy ta lista zadań, omawiane będą na kolejnych wykładach. Przed rozpoczęciem pracy nad zadaniami warto wcześniej zapoznać się z rozdziałami 3, 4 oraz 5.1 w podręczniku *Algorithms* S. Dasgupty et al. [2]. W podręczniku *Introduction to Algorithms* Th. Cormena et al. [1] tematyce algorytmów grafowych poświęcona jest cała część VI, obejmująca rozdziały od 22 do 26. W obu tych książkach znajdą Państwo również przykładowe pseudokody algorytmów, które należy zaimplementować i przetestować.

Zadanie dodatkowe (**Zadanie 5.***) zostanie opublikowane jako osobna lista zadań (lista 5D). Literatura do zadania dodatkowego podana zostanie w jego opisie.

Zadanie 1. [15%]

Napisz program, który implementuje działanie kolejki priorytetowej. Struktura powinna umożliwiać przechowywanie wartości typu `int` wraz z ich priorytetem, będącym nieujemną liczbą całkowitą, przy czym najwyższym priorytetem jest wartość 0. Program nie powinien wymagać żadnych parametrów uruchomienia, a na standardowym wejściu przyjmuje dodatnią liczbę całkowitą M (liczba operacji), a następnie (w liniach od 2 do $M + 1$) jedną z operacji:

- **insert** x p – wstaw do struktury wartość x o priorytecie p,
- **empty** – wypisz wartość 1 (`true`) dla pustej struktury, wartość 0 (`false`) w p.p.
- **top** – wypisz wartość o najwyższym priorytecie lub pustą linię w przypadku braku elementów w strukturze,
- **pop** – wypisz wartość o najwyższym priorytecie, a następnie usuń ją ze struktury (wypisz pustą linię w przypadku braku elementów w strukturze),
- **priority** x p – dla każdego elementu o wartości x obecnego w strukturze ustawia priorytet p, jeśli jest on wyższy od aktualnego priorytetu danego elementu
- **print** – wypisuje w jednej linii zawartość struktury w postaci $(x_i,\, p_i)$, gdzie x_i to kolejne wartości przechowywane w kolejce, a p_i to odpowiadające im priorytety.

Dla liczby elementów w kolejce n koszt operacji musi wynosić $O(\log n)$ dla wszystkich poleceń za wyjątkiem **print**. Przetestuj implementację na wybranych przez siebie przykładowych danych, np. dla tablicy $n = 1000$ liczb całkowitych (z możliwymi powtórzeniami) wraz z ich priorytetami (np. wybieranymi losowo z ustalonego zakresu od 0 do $P < n$).

Krótka charakterystyka wybranych implementacji kolejek priorytetowych podana jest w rozdziale 4.5 w [2] (str. 125–127). Kopce binarne i bazujące na nich kolejki priorytetowe omówione są w rozdziale 6 w [1]. Nieco bardziej zaawansowana struktura danych, nazywana kopcem Fibonacciego, omówiona jest w rozdziale 19 w [1].

Zadanie 2. [25%]

Korzystając ze struktury zaimplementowanej w **Zadaniu 1.** lub kopca Fibonacciego, zaimplementuj program realizujący algorytm Djikstry, który dla podanego grafu skierowanego $G = (V, E)$ znajduje najkrótsze ścieżki z wybranego wierzchołka $\tilde{v} \in V$ do każdego $v \in V$.

Program nie powinien wymagać żadnych parametrów wywołania. Po uruchomieniu programu, na standardowym wejściu podajemy definicję grafu G oraz wierzchołek startowy $\tilde{v}$. Kolejno wczytywane są:

- liczba wierzchołków $n = |V|$ (przyjmujemy, że wierzchołki są etykietowane kolejnymi liczbami naturalnymi ze zbioru $\{1, \ldots n\}$),
- liczba krawędzi $m = |E|$ (krawędzie są postaci (u, v, w), gdzie u to źródło krawędzi, v jest wierzchołkiem docelowym, a w – wagą krawędzi; zakładamy, że wagi są nieujemnymi liczbami rzeczywistymi, ale niekoniecznie spełniona jest nierówność trójkąta; ponadto przyjmujemy, iż ścieżka z u do u zawsze istnieje i ma koszt 0),
- kolejno m definicji krawędzi w postaci `u v w`,
- etykieta wierzchołka startowego $\tilde{v}$.

Na standardowym wyjściu powinno zostać wypisane n linii, w formacie `id_celu waga_drogi`, natomiast na standardowym wyjściu błędów powinny być wypisane dokładne ścieżki (tzn. wierzchołki pośrednie i wagi) do każdego z wierzchołków docelowych oraz czas działania programu w milisekundach.

Przetestuj swoją implementację algorytmu Dijkstry. Przykładowe dane testowe (definicja grafu bez etykiety wierzchołka startowego) znajdują się w dołączonych plikach `g8.txt`, `g250.txt`, `g1000.txt` oraz `g10000.txt`.

Algorytm Dijkstry (wraz z pseudokodem oraz najważniejszymi własnościami) omówiony jest w rozdziale 4.4 w [2] oraz w rozdziale 24.3 w [1].

Zadanie 3. [35%]

Korzystając ze struktury z **Zadania 1**, zaimplementuj algorytmy znajdujące dla podanego nieskierowanego grafu spójnego $G = (V, E)$ minimalne drzewo rozpinające. Program powinien umożliwiać wykonanie algorytmu Prima (parametr wywołania `-p`) oraz algorytmu Kruskala (parametr wywołania `-k`). Niezależnie od parametru uruchomienia, dane wejściowe przyjmują postać:

- liczba wierzchołków $n = |V|$ (przyjmujemy, że wierzchołki są etykietowane kolejnymi liczbami naturalnymi ze zbioru $\{1, \ldots, n\}$),
- liczba krawędzi $m = |E|$ (krawędzie są postaci $(\{u, v\}, w)$, gdzie $u \in V$ i $v \in V$ są połączonymi wierzchołkami, a w – wagą krawędzi; zakładamy, że wagi są nieujemnymi liczbami rzeczywistymi, ale niekoniecznie spełniona jest nierówność trójkąta; ponadto przyjmujemy, że krawędź z u do u zawsze istnieje i ma koszt 0),
- kolejno m definicji krawędzi w postaci `u v w`.

Na standardowym wyjściu w kolejnych liniach powinny zostać wypisane krawędzie `u v w` tworzące drzewo rozpinające (wypisując krawędzie nieskierowane przyjmujemy konwencję, że $u < v$) oraz łączna waga wyznaczonego drzewa rozpinającego.

Przetestuj swoje implementacje algorytmów Prima i Kruskala. Możesz wygenerować własne dane testowe (kilka grafów różnych rozmiarów, od kilku do kilkudziesięciu tysięcy wierzchołków) lub stworzyć je w oparciu o przykładowe dane testowe z **Zadania 2.** (pamiętaj, że grafy z poprzedniego zadania były skierowane, tj. rozróżnialiśmy krawędzie (u, v, w) z u do v oraz (v, u, w) z v do u; w tym zadaniu rozważamy grafy nieskierowane).

Problem wyznaczania minimalnego drzewa rozpinającego (ang. *minimum spanning tree, MST*) oraz klasyczne algorytmy Prima i Kruskala omówione są w rozdziale 5.1 w [2] i w rozdziale 23 w [1].

Zadanie 4. [25%]

Rozważmy pełny graf nieskierowany (klikę) $K_n = (V_n, E_n)$ o n wierzchołkach, gdzie $V_n = \{1, \ldots, n\}$ jest zbiorem wierzchołków, a $E = \{(\{u, v\}, w) \colon u, v \in V \wedge u \neq v \wedge w \in \mathbb{R}_+\}$ zbiorem krawędzi z dodatnimi wagami (mamy $|E| = m = \frac{n(n-1)}{2}$). Zakładamy, że wagi krawędzi **spełniają** nierówność

trójkąta. Zaimplementuj algorytmy, w których startując z dowolnego wierzchołka, trawersujemy krawędzie do czasu odwiedzenia wszystkich wierzchołków, zgodnie z następującymi strategiami.

1. Proste błądzenie losowe (ang. *simple random walk*) – w każdym kolejnym kroku, znajdując się w wierzchołku $u \in V$, losujemy niezależnie z jednakowym prawdopodobieństwem równym $\frac{1}{n-1}$ jedną z $n-1$ krawędzi, którą następnie przechodzimy do kolejnego wierzchołka. Jeśli wylosowana krawędź to $(\{u, v\}, w)$, to w kolejnym kroku znajdujemy się w wierzchołku v, gdzie powtarzamy całą procedurę.

2. Strategia zachłanna – będąc w wierzchołku $u \in V$, wybieramy krawędź $(\{u, v\}, w)$ o najniższej wadze spośród krawędzi prowadzących do wciąż nieodwiedzonych wierzchołków. Następnie przechodzimy do wierzchołka v i powtarzamy procedurę.

3. Wykorzystując algorytmy z **Zadania 3**, budujemy minimalne drzewo rozpinające T, a następnie na jego podstawie konstruujemy *cykl Hamiltona* w klice K_n, tj. taki cykl, w którym każdy wierzchołek (oprócz początkowego) odwiedzany jest dokładnie raz. Eksploracja kliki polega na przechodzeniu kolejnych krawędzi wyznaczanego cyklu Hamiltona (za wyjątkiem ostatniej – nie wymagamy powrotu do wierzchołka, w którym zaczynaliśmy eksplorację).

   Cykl ten możemy otrzymać w następujący sposób – dublujemy każdą z krawędzi w drzewie T, tworząc graf T^*, który jest grafem eulerowskim (tzn. grafem, w którym istnieje *cykl Eulera* – taki cykl, który przechodzi przez każdą krawędź dokładnie raz). Następnie wyznaczamy w grafie T^* cykl Eulera. Eksplorując graf K_n, przechodzimy ścieżką wyznaczoną przez pierwsze wystąpienie każdego z wierzchołków ze zbioru V na cyklu Eulera w T^*. Innymi słowy, usuwamy ponowne wystąpienia wszystkich wierzchołków na cyklu Eulera w T^*, trawersując krawędzie grafu K_n wyznaczone przez kolejne pary pozostałych wierzchołków.

   Zaimplementuj powyższą strategię w możliwie efektywny sposób (patrz np. rozdział 35.2.1 w [1]). Sprawdź czy drzewa zbudowane przy użyciu algorytmu Prima dają zawsze te same wyniki co zbudowane przy użyciu algorytmu Kruskala.

Na standardowym wejściu podawane są kolejno:

- liczba wierzchołków $n = |V|$,
- m definicji krawędzi w postaci `u v w`.

Wybór wierzchołka startowego powinien być dokonywany przez algorytm (np. wybór losowego lub ustalonego wierzchołka).

Na standardowym wyjściu powinny zostać wypisane trzy linie (kolejno dla strategii błądzenia losowego, wyboru krawędzi o najniższej wadze, minimalnego drzewa rozpinającego) postaci: `k W M t`, gdzie `k` oznacza liczbę wykonanych kroków podczas odwiedzania wierzchołków, `W` – łączny koszt trasy, `M` – zużytą dodatkową pamięć (np. na pamiętanie odwiedzonych już wierzchołków), a `t` – czas działania algorytmu (wraz z preprocessingiem, np. budowaniem drzewa). Na standardowym wyjściu błędów powinny być drukowane pełne trasy wykonane dla każdej strategii.

Wykonaj testy dla grafów K_n, gdzie $n \in \{5,\ 50,\ 500,\ 5\,000,\ 50\,000,\ 500\,000\}$. Przy generowaniu wag krawędzi pamiętaj, aby spełniały one nierówność trójkąta. W przypadku błądzenia losowego wykonaj K niezależnych powtórzeń (np. $K = 20/50/100$) i dodatkowo wyznacz średnią liczbę kroków $\overline{k}$ oraz średni łączny koszt trasy $\overline{W}$ dla każdego n. Spróbuj wyznaczyć asymptotyczne oszacowanie średniej liczby kroków wykonywanych przez błądzenie losowe.

Dodatkowe uwagi i literatura Zagadnienie eksploracji grafów jest jednym z fundamentalnych, intensywnie badanych problemów algorytmicznych. W zależności od rozważanych scenariuszy i dostępnych zasobów, istnieje szereg różnych algorytmów dla problemu eksploracji, które znajdują liczne zastosowania. Jednymi z elementarnych, klasycznych technik są algorytmy przeszukiwania grafów wgłąb (ang. *depth-first search, DFS*) oraz wszerz (ang. *breadth-first search, BFS*), por. rozdziały 3.2, 3.3 i 4.2 w [2] oraz 22.2 i 22.3 w [1]. Algorytmy te zostaną szczegółowo omówione na wykładzie.

W zadaniu dodatkowym (**Zadanie 5.*** z listy 5D) pokazane zostaną przykłady prostych technik eksploracji nieznanego grafu sieci przez mobilnego agenta w środowisku rozproszonym.

Strategie z punktu 2. (strategia zachłanna) i 3. (algorytm bazujący na MST) są ściśle związane z innym bardzo znanym problemem algorytmicznym dla grafów, nazywanym *problemem komiwojażera* (ang. *traveling salesman problem, TSP*), por. np. rozdział 35.2 w [1] lub 8.1 w [2]. Problem ten polega na znalezieniu – dla zadanego grafu pełnego K_n o n wierzchołkach z dodatnimi wagami krawędzi – cyklu Hamiltona o najmniejszym koszcie (zdefiniowanym jako suma wag krawędzi tworzących ten cykl). TSP jest problemem NP-zupełnym, co w praktyce oznacza, że nie spodziewamy się istnienia efektywnego algorytmu (tj. deterministycznego algorytmu działającego w czasie wielomianowym względem n) rozwiązującego ten problem. Strategia 2. dostarcza pewnej heurystyki, aczkolwiek w ogólności nie gwarantuje ona uzyskania rozwiązania bliskiego optymalnemu. W przypadku, gdy wagi krawędzi grafu spełniają nierówność trójkąta, algorytm 3. jest *algorytmem $\frac{1}{2}$-aproksymacyjnym*, tj. zwrócone rozwiązanie (cykl Hamiltona) ma co najwyżej 2 razy większy koszt niż rozwiązanie optymalne. Pewna modyfikacja strategii 3, znana jako *algorytm Christofidesa*, jest algorytmem $\frac{1}{3}$-aproksymacyjnym, tj. gwarantuje uzyskanie rozwiązania co najwyżej $\frac{3}{2}$ razy gorszego od optimum (o ile wagi krawędzi spełniają nierówność trójkąta). Elementy teorii złożoności obliczeniowej (w ujęciu nieco bardziej algorytmicznym) obejmujące charakterystykę klas P oraz NP i omówienie wybranych problemów NP-zupełnych można znaleźć w rozdziale 34 w [1] oraz w rozdziale 8 w [2]. Tematy te zostaną szczegółowo omówione na kursie *Teoria obliczeń i złożoność obliczeniowa* (II stopień, specjalność Algorytmika; na studiach I stopnia elementy złożoności obliczeniowej pojawiają się na kursie wybieralnym *Teoretyczne podstawy informatyki*).

Błądzenie losowe na grafach jest jednym z fundamentalnych przykładów łańcuchów Markowa – bardzo ważnej klasy procesów stochastycznych. Elementy teorii łańcuchów Markowa oraz ich zastosowanie w informatyce do projektowania oraz badania algorytmów dla szeregu problemów obliczeniowych przedstawione są m.in. w [3, 4, 5].

Literatura

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[2] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh Vazirani. *Algorithms*. McGraw-Hill, Inc., USA, 1st edition, 2006.

[3] Olle Häggström. *Finite Markov Chains and Algorithmic Applications*. Cambridge University Press, 3rd edition, 2002.

[4] David A. Levin, Yuval Peres, and Elizabeth L. Wilmer. *Markov Chains and Mixing Times*. American Mathematical Society, 1st edition, 2009.

[5] Michael Mitzenmacher and Eli Upfal. *Probability and Computing. Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.