Projekt 2

Matematyka dyskretna

Zastosowania przeszukiwania grafów

Ivan Kaliankovich

Spis treści

Algorytm przeszukiwania grafu w głąb

Funkcja DFS przeszukiwania w głąb i wypisywania wszystkich odwiedzonych wierzchołków:

```
function odwiedzone = dfs(G, v)
    odwiedzone = [];
    stos = [v];
    while ~isempty(stos)
        u = stos(1);
        stos = stos(2:end);
        if u >= 1 && ~any(ismember(odwiedzone, u))
            odwiedzone = [odwiedzone u];
            fprintf('%d\n', u);
            for v = find(G(u, :))
                if ~ismember(v, odwiedzone)
                    stos = [stos v];
                    fprintf('%d -> %d\n', u, v);
                end
            end
        end
    end
end
```

Funkcja polega na użyciu stosu

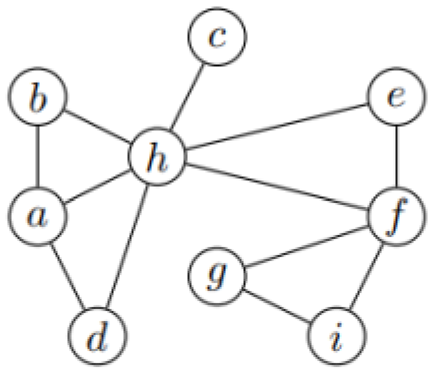
Algorytm przeszukiwania grafu we wszerz

Funkcja BFS przeszukiwania we wszerz i wypisywania wszystkich odwiedzonych wierzchołków:

```
function odwiedzone = bfs(G, v)
    odwiedzone = [];
    kolejka = [v];
    while ~isempty(kolejka)
        u = kolejka(1);
        kolejka = kolejka(2:end); % Usunięcie pierwszego elementu kolejki
        if ~ismember(u, odwiedzone)
            odwiedzone = [odwiedzone u];
            fprintf('%d\n', u);
            for v = find(G(u, :))
                if ~ismember(v, odwiedzone)
                    fprintf('%d -> %d\n', u, v);
                    kolejka = [kolejka v];
                end
            end
        end
    end
end
```

Funkcja polega w tym przypadku na użycie kolejki

Zad 1



Dla powyższego grafu została stworzona macierz sąsiedztwa

```
G = [
    0 1 0 1 0 0 0 1 0
    1 0 0 0 0 0 0 1 0
    0 0 0 0 0 0 0 1 0
    1 0 0 0 0 0 0 1 0
    0 0 0 0 0 1 0 1 0
    0 0 0 0 1 0 1 1 1
    0 0 0 0 0 1 0 0 1
    1 1 1 1 1 1 0 0 0
    0 0 0 0 0 1 1 0 0
];
```

Kolejne wierzchołki od a do i odpowiadają wierszom, natomiast kolumny oznaczają czy dany wierzchołek ma krawędź z innym wierzchołkiem.
Po wywołaniu funkcji DFS przez graf G i punkt początkowy a dostaję taką odpowiedź :

```
WYPISYWANIE DFS
1
1 -> 2
1 -> 4
1 -> 8
2
2 -> 8
4
4 -> 8
8
8 -> 3
8 -> 5
8 -> 6
3
5
5 -> 6
6
6 -> 7
6 -> 9
7
7 -> 9
9
```

Natomiast funkcja BFS:

```
WYPISYWANIE BFS
1
1 -> 2
1 -> 4
1 -> 8
2
2 -> 8
4
4 -> 8
8
8 -> 3
8 -> 5
8 -> 6
3
5
5 -> 6
6
6 -> 7
6 -> 9
7
7 -> 9
9
```

Zad 2

Cel zadania

Celem tego zadania było znalezienia najtańszych połączeń samolotowych dla turystów, którzy nie chcą zatrzymywać się w jednym miejscu, lecz chcą mieć długie podróże zawierające dużo przelotów i przystanków z tym warunkiem, że żaden z przystanków nie może się powtarzać. Przystanki / punkty do odwiedzenia są w tym przypadku wierzchołkami grafu oraz loty są krawędziami pomiędzy tymi punktami.

Wejście / wyjście

Na wejście podaje się macierz sąsiedztwa, czyli punkty turystyczne z wagami oraz największy koszt, który turysta jest w stanie ponieść. Na wyjście uzyskuję się połączenia drzewa, które zostało uzyskane z grafu podanego na wejście z tym warunkiem, że nigdy nie zostanie przekroczona kwota maksymalna.

Narzędzie do realizacji zadania

Do rozwiązania zadania został użyty algorytm Kruskala. Jest dobrym do tego narzędziem, ponieważ algorytm Kruskala sortuję krawędzi grafu według ich wag. Sortowanie pozwala algorytmowi wybrać krawędzie o najmniejszej wadze w pierwszej kolejności, co jest celem turysty.

Realizacja zadania

Do realizacji zadania stworzona została funkcja Kruskal

```
function [minimalneDrzewo, koszt] = kruskal(macierzSasiedztwa, maksymalnyKoszt)
    n = size(macierzSasiedztwa, 1); % liczba miast
    krawedzie = [];

    % Tworzenie listy krawędzi w formie [waga, miasto1, miasto2]
    for i = 1:n
        for j = i+1:n
            if macierzSasiedztwa(i, j) > 0
                krawedzie = [krawedzie; macierzSasiedztwa(i, j), i, j];
            end
        end
    end

    % Sortowanie krawędzi według wag
    krawedzie = sortrows(krawedzie);

    % Inicjalizacja zbiorów rozłącznych
    rodzice = 1:n;

    minimalneDrzewo = {};
    koszt = 0;

    % Algorytm Kruskala
    for i = 1:size(krawedzie, 1)
        waga = krawedzie(i, 1);
        miasto1 = krawedzie(i, 2);
        miasto2 = krawedzie(i, 3);

        if check_cykl(rodzice, miasto1) ~= check_cykl(rodzice, miasto2) % Sprawdzenie, czy cykl
            if koszt + waga <= maksymalnyKoszt % Sprawdzenie, czy przekroczony max koszt
                koszt = koszt + waga;
                minimalneDrzewo{end+1} = [miasto1, miasto2];

                % Połączenie zbiorów rozłącznych
                rodzice(check_cykl(rodzice, miasto1)) = check_cykl(rodzice, miasto2);
            else
                break; % Przerwanie algorytmu, gdy osiągnięto maksymalny koszt
            end
        end
    end
end
```

Funkcja przyjmuje na wejście macierz sąsiedztwa oraz max koszt, na wyjście zwraca koszt wszystkich krawędzi, które znalazł algorytm.

```
function reprezentant = check_cykl(rodzice, miasto)
    while rodzice(miasto) ~= miasto
        miasto = rodzice(miasto);
    end
    reprezentant = miasto;
end
```

Algorytm też korzysta z funkcji pomocniczej do sprawdzenia czy wybierając krawędź, czyli zbiór wierzchołków zostanie osiągnięty cykl w grafie. Jeśli mają różnych reprezentantów, oznacza to, że należą do różnych zbiorów rozłącznych, i można je połączyć bez utworzenia cyklu w grafie minimalnego drzewa rozpinającego.

```
miastaWloch = [
    0 570 220 700 0 510 300 40 340;    % 0 - Rome
  570 0 760 400 0 150 350 0 0;    % 1 - Milano
  220 760 0 830 0 400 0 670 0;    % 2 - Napoli
  700 400 830 0 0 630 0 0 520;    % 3 - Torino
    0 0 0 0 0 0 0 0 0;            % 4 - Palermo (brak połączeń w przykładowej macierzy)
  510 150 400 630 0 0 250 0 0;    % 5 - Genova
    300 350 0 0 0 250 0 900 0;      % 6 - Bologna
    40 0 670 0 0 0 900 0 620;      % 7 - Catania
    340 0 0 520 0 0 0 620 0         % 8 - Venezia
];

[minDrzewo, minKoszt] = kruskal(miastaWloch, 1000);
disp(minDrzewo);
disp(minKoszt);
```

Do sprawdzenia wyników utworzyłem macierz o nazwie miasta Włoch posiadającej 9 elementów i podałem na wejście . Uzyskałem następujące wyniki:

```
>> zad2
    {[1 8]}    {[2 6]}    {[1 3]}    {[6 7]}    {[1 7]}

   960
```

Podsumowanie

Algorytm Kruskala doskonale się sprawdza do znalezienia drzewa rozpinającego w grafie o najmniejszej wadzę. W podanym przeze mnie przykładzie znalazł najlepsze połączenia, natomiast decyzję o trasie podejmuję turysta bazując na przeznaczonym przez niego budżecie oraz o wynik algorytmu. Widząc krawędzie wyniku algorytmu Kruskala jest w stanie stworzyć manualnie trasę podróży.