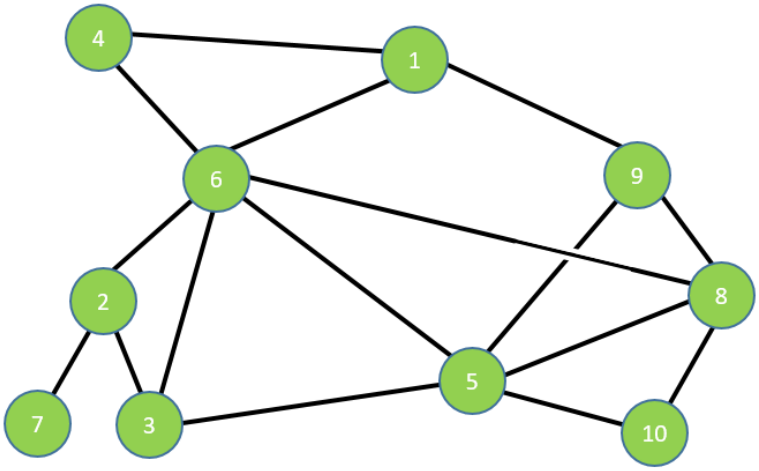
Wszerz, czyli którędy najbliżej?

#grafy #wierzchołki
#krawędzie #listy_sąsiedztwa
#BFS

W tym podrozdziale rozpoczynamy przegląd podstawowych algorytmów grafowych. Samo pojęcie grafu pojawia się w niezliczonych sytuacjach, na przykład gdy spojrzymy na mapę samochodową: mamy na niej miejscowości, czyli *wierzchołki grafu* (ang. **vertex**, liczba mnoga: **vertices**, wymawiaj: *wertises*, z akcentem na pierwsze *e*) oraz łączące je drogi, czyli *krawędzie grafu* (ang. **edge**, wymawiaj: *edż*; liczba mnoga: **edges**, wymawiaj: *edżes*). Oto przykładowy graf nieskierowany, czyli taki, którego krawędzie nie mają określonego zwrotu:



Wymieńmy rzucające się w oczy cechy tego grafu: dowolne dwa wierzchołki są połączone co najwyżej jedną krawędzią, zaś krawędzie nie mają punktów wspólnych poza wierzchołkami. Ponadto nie ma krawędzi zaczynających się i kończących w jednym i tym samym wierzchołku, a z każdego wierzchołka można dojść do dowolnego innego (choć być może trzeba przejść przez kilka krawędzi).*

Mamy tutaj jedynie wierzchołki (ponumerowane od 1 do 10) i łączące je krawędzie, natomiast w praktycznych problemach wierzchołkom i krawędziom mogą być przypisane pewne dane. Będziemy rozwiązywać takie zadania w przyszłości.

Najważniejsze parametry charakteryzujące graf to liczba wierzchołków V oraz liczba krawędzi E. Jeśli graf spełnia opisane wyżej warunki, wtedy mamy $V - 1 \leqslant E \leqslant \frac{1}{2}V(V-1)$ (dlaczego?).

Pozostaje kwestia, w jaki sposób reprezentować graf w kodzie programu? Generalnie rzecz biorąc, wystarczy zapamiętać, co jest czego sąsiadem (mówimy o wierzchołkach grafu). Jednak

*Dla grafu nieskierowanego ta ostatnia cecha nazywana jest *spójnością* – mówiąc kolokwialnie chodzi o to, że graf jest w jednym kawałku.

zbiór krawędzi trzeba jakoś uporządować, aby operacje na nim odbywały się względnie szybko. Jedna z metod polega na tym, że dla każdego wierzchołka zapamiętujemy jego listę sąsiadów. Jest to tak zwana *lista sąsiedztwa* (ang. **adjacency list**, wymawiaj: *adżasensy list*, z akcentem na drugie *a*).[†] Na przykład listę sąsiedztwa wierzchołka 1 stanowią wierzchołki 4, 6, 9 – niekoniecznie w tej kolejności.

Każda lista sąsiedztwa może być innej długości, zatem narzuca się użycie jakiejś elastycznej struktury – zwykle jest to wektor liczb całkowitych. Kluczowy jest jednak format danych wejściowych – zwykle najpierw podaje się wartości V (ilość wierzchołków) oraz E (ilość krawędzi), a potem następuje E wierszy zawierających po dwie liczby całkowite z zakresu od 1 do V – są to numery wierszy połączonych krawędzią:

$V \quad E$
$a_0 \quad b_0$
$a_1 \quad b_1$
$\ldots$
$a_{E-1} \quad b_{E-1}$

Dla grafu nieskierowanego kolejność liczb w wierszu jest obojętna, bo skoro istnieje krawędź z wierzchołka a do wierzchołka b, to ta sama krawędź prowadzi od wierzchołka b do wierzchołka a. Z kolei kolejność wierszy z opisem krawędzi ma wpływ na kolejność wierzchołków na listach sąsiedztwa – na ogół jest to bez większego znaczenia (no, chyba że są pewne przesłanki co do tej kolejności).

Listy sąsiedztwa trzeba zebrać w jedną strukturę – również wektor, którego elementami są poszczególne listy sąsiedztwa. Każdy element tego zbiorczego wektora (nazwiemy go G) odpowiada jednemu wierzchołkowi i jego liście sąsiedztwa. Ponieważ wierzchołki grafu są z reguły numerowane od 1 do V, dlatego też zadelarujemy wektor G o rozmiarze $V + 1$, a jego zerowego elementu nie będziemy wykorzystywać.

Wczytywanie grafu

Zaczniemy od wczytania danych wejściowych spełniających powyższą specyfikację. Dla naszego przykładowego grafu dane mogą wyglądać na przykład tak:

```
10 16
4 1
1 6
6 4
2 6
3 6
2 7
3 5
6 5
10 5
8 5
9 5
9 1
```

[†]Alternatywnym sposobem jest użycie dwuwymiarowej bitowej *tablicy sąsiedztwa*, ale nie będziemy się tutaj tym zajmować.

```
6 8
2 3
8 9
8 10
```

Napiszemy funkcję `read_graph()`, która wczyta powyższe dane umieszczając je w odpowiednich zmiennych:

```
void read_graph(vector<vector<int>> &G, int &V, int &E)
{
  cin >> V >> E;
  G.resize(V + 1);
  . . .
}
```

Teraz G to wektor złożony z $V + 1$ pustych wektorów i możemy wczytać opis E krawędzi. Jeśli mamy krawędź a—b, to znaczy, że na liście sasiedztwa wierzchołka a trzeba dopisać wierzchołek b, oraz na liście sąsiedztwa wierzchołka b dopisać wierzchołek a:

```
for(int i = 0; i < E; i++)
{
  int a, b;
  cin >> a >> b;
  G[a].push_back(b);
  G[b].push_back(a);
}
```

Jeśli chodzi o funkcję `read_graph()`, to w zasadzie finito. Wypada jeszcze napisać ją w całości, *le voilà* (wymawiaj: *le wuala*, z akcentem na drugie *a*):[‡]

```
void read_graph(vector<vector<int>> &G, int &V, int &E)
{
  cin >> V >> E;
  G.resize(V + 1);
  for(int i = 0; i < E; i++)
  {
    int a, b;
    cin >> a >> b;
    G[a].push_back(b);
    G[b].push_back(a);
  }
}
```

Możemy jeszcze napisać taką kontrolną funkcję `print_graph()`, która po wczytaniu danych wejściowych wypisze listy sąsiedztwa wszystkich wierzchołków. Lista sąsiedztwa wierzchołka v to po prostu $G[v]$ i jej zawartość należy wypisać.

[‡] *Oto jest* (w języku Astérixa).

Tak prezentuje się funkcja `print_graph()`:

```
void print_graph(vector<vector<int>> &G, int V)
{
  for(int v = 1; v <= V; v++)
  {
    cout << v << ": ";
    for(int x : G[v])
      cout << x << " ";
    cout << endl;
  }
}
```

Sklecimy sobie niewielki program do testów:

```
#include <bits/stdc++.h>
using namespace std;

void read_graph(vector<vector<int>> &G, int &V, int &E)
{
  . . .
}

void print_graph(vector<vector<int>> &G, int V)
{
  . . .
}

int main()
{
  int V, E;
  vector<vector<int>> G;
  read_graph(G, V, E);
  print_graph(G, V);
}
```

Po jego uruchomieniu i wklepaniu danych opisujących nasz graf otrzymujemy następujący wynik:

```
1: 4 6 9
2: 6 7 3
3: 6 5 2
4: 1 6
5: 3 6 10 8 9
6: 1 4 2 3 5 8
7: 2
8: 5 6 9 10
9: 5 1 8
10: 5 8
```

Przeszukiwanie grafu wszerz

W tej sekcji opiszemy najbardziej fundamentalny algorytm przeszukiwania grafu: przeszukiwanie grafu wszerz, czyli **BFS** (ang. **Breadth-First Search**, wymawiaj: *bredζ ferst sercz*)[§]

Ogólnie rzecz biorąc, trzeba umieć graf *przeszukać*, czyli przespacerować się po wszystkich jego wierzchołkach, w miarę możliwości robiąc to szybko i sprawnie. W szczególności pasowałoby wizytować każdy wierzchołek tylko raz, a to oznacza, że trzeba zaznaczać, że było się w danym miejscu.[¶]

W zależności od wybranej metody przeszukiwania (i rodzaju grafu) można liczyć na rozmaite profity z takiego spaceru (poza satysfakcją). W szczególności można (czasem wręcz trzeba) zapamiętać trasę, jaką dotarliśmy w dane miejsce i/lub odległość od punktu, w którym zaczęliśmy wędrówkę.[‖]

Każde przeszukiwanie musi rozpocząć się w określonym wierzchołku – oznaczymy go przez s (jak **start**):

```
void BFS(vector<vector<int>> G, int s,
         vector<bool> &visited, vector<int> &parent, vector<int> &dist)
{

}
```

Parametry G oraz s już rozpoznajemy, a pozostałe trzy wektory mają następujące znaczenie:

- *visited* – tu zapisujemy, czy dany wierzchołek był już odwiedzony (musi być zainicjalizowany wartością `false`),
- *parent* – dla każdego odwiedzonego wierzchołka tu zapisany będzie numer bezpośredniego poprzednika, czyli wierzchołka, z którego dostaliśmy się w to miejsce (inicjalizowane zerami – neutralną wartością),
- *dist* (ang. **distance**) – odległość od wierzchołka s (nie trzeba inicjalizować).

Zaczynamy od odwiedzenia wierzchołka s (w domyśle jest on nieodwiedzony) i ustawieniu $dist[s]$ na zero (odległość s od s):

```
visited[s] = true;
dist[s] = 0;
```

Wartość $parent[s]$ pozostaje równa 0 – wierzchołek startowy jest *sierotą*.

Następnie deklarujemy kolejkę, do której dokładać będziemy kolejne odwiedzane wierzchołki (zaczniemy od wierzchołka startowego).

[§]Symbol ζ oznacza seplenione *z*.

[¶]Tak jak światli turyści piszą na zabytkach Krakowa: *Byłem tu! Pioter z Wąchocka*. Niektórzy nawet datę wyskrobują – przyda nam się ten zwyczaj podczas przeszukiwania grafu w głąb.

[‖]Standardowo w przypadku grafów, dla których nie określono wag krawędzi, przyjmuje się, że wszystkie krawędzie mają taką samą wagę (koszt, długość, whatever) równą 1. Ta sama zasada obowiązuje dla wierzchołków grafu.

Przetwarzanie kolejki będzie odbywać się, póki nie stanie się ona pusta:

```
queue<int> Q;
Q.push(s);
while(!Q.empty())
{
  . . .
}
```

Co piszczy w pętli? Najpierw zdejmujemy z czoła kolejki znajdujący się tam wierzchołek i zabieramy się do przeglądania listy jego sąsiadów:

```
int v = Q.front();
Q.pop();
for(int x : G[v])
  . . .
```

Interesują nas tylko wierzchołki nieodwiedzone – każdy taki wierzchołek oznaczamy jako odwiedzony, ustawiamy jego *parent*-a na wierzchołek v, a odległość na $dist[v]+1$, po czym wstawiamy go do kolejki:

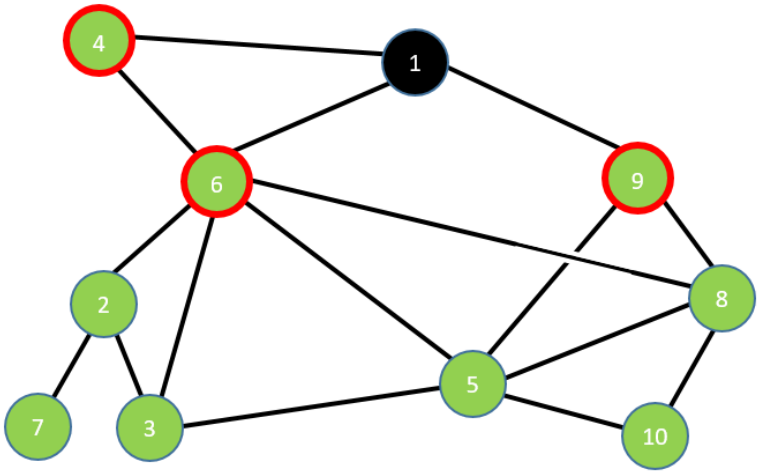
```
if(!visited[x])
{
  visited[x] = true;
  parent[x] = v;
  dist[x] = dist[v] + 1;
  Q.push(x);
}
```

Cała funkcja `BFS()` prezentuje się następująco:

```
void BFS(vector<vector<int>> G, int V, int s,
         vector<bool> &visited, vector<int> &parent, vector<int> &dist)
{
  visited[s] = true;
  dist[s] = 0;
  queue<int> Q;
  Q.push(s);
  while(!Q.empty())
  {
    int v = Q.front();
    Q.pop();
    for(int x : G[v])
      if(!visited[x])
      {
        visited[x] = true;
        parent[x] = v;
        dist[x] = dist[v] + 1;
```

```
            Q.push(x);
          }
        }
      }
```

Prześledzimy działanie tego algorytmu na naszym grafie dla $s = 1$. Kolorem czarnym zaznaczymy wierzchołki odwiedzone, a czerwoną obwódką – wierzchołki właśnie rozpatrywane:



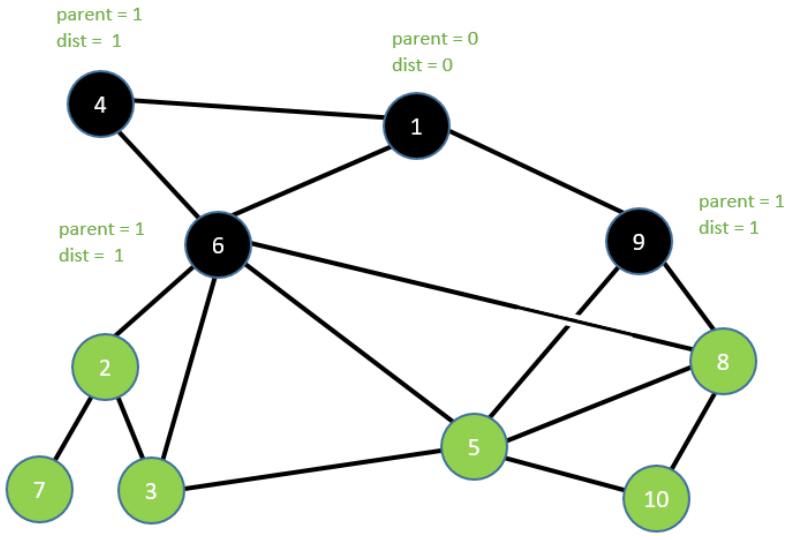
Na początku w kolejce mamy tylko wierzchołek 1, którego numer zdejmujemy i umieszczamy w zmiennej v. (Kolejka Q chwilowo jest pusta, ale zaraz pojawią się w niej nowe wierzchołki.) Teraz rozpatrujemy jego listę sąsiedztwa:

$4, 6, 9$

Trzymamy się kolejności, w jakiej wierzchołki wyświetlone były przez funkcję `print_graph()` w poprzedniej sekcji.

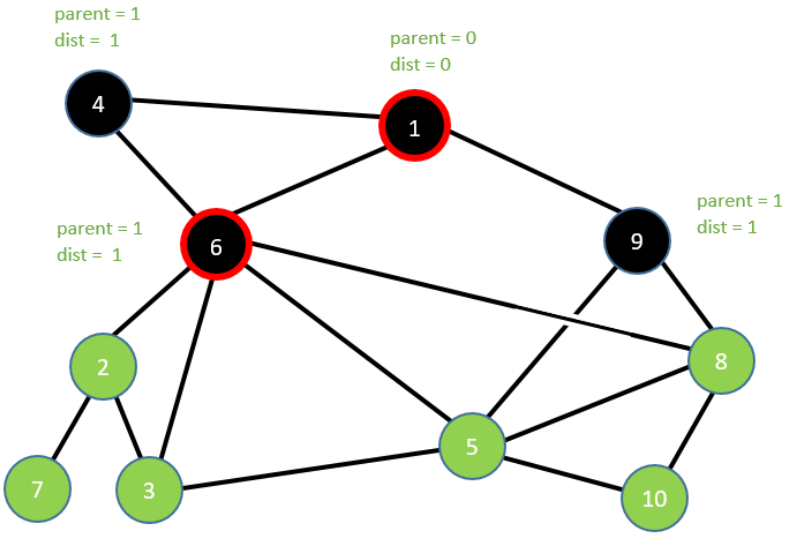
Interesują nas tylko wierzchołki nieodwiedzone – każdy taki wierzchołek oznaczamy jako odwiedzony, ustawiamy jego *parent*-a na wierzchołek v, a odległość na $dist[v] + 1$, po czym wstawiamy go do kolejki.

Tym razem nieodwiedzone będą wszystkie wierzchołki z listy, zatem po wykonaniu pętli `for` mamy taką sytuację:



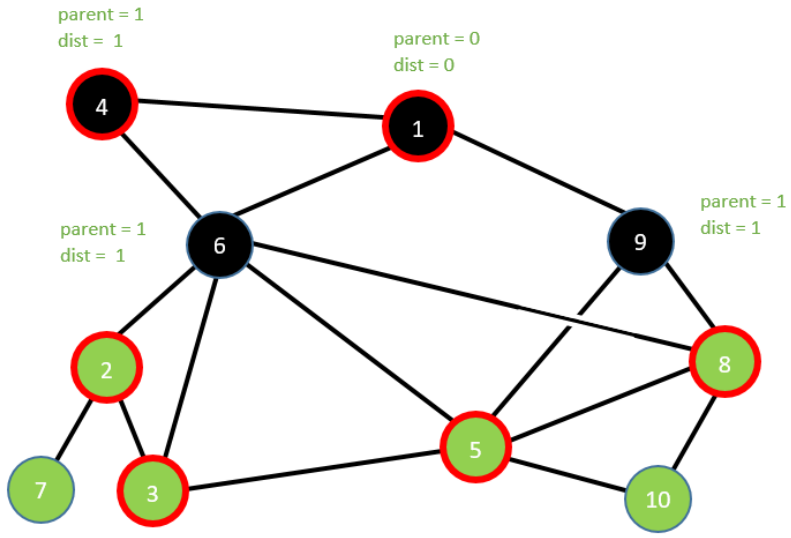
zaś w kolejce znajdują się wierzchołki 4, 6, 9 – właśnie w tej kolejności.

Zaczyna się kolejne obejście pętli `while`: z czoła kolejki zdejmujemy 4 i podstawiamy jako wartość zmiennej v. Przeglądamy listę sąsiadów tego wierzchołka:

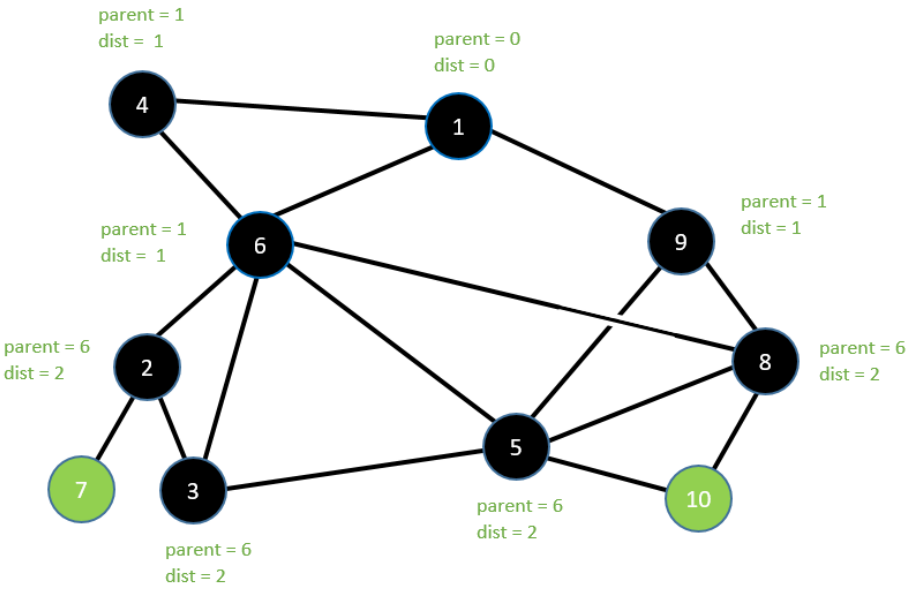


Mamy tu wierzchołki 1 oraz 6 – obydwa już odwiedzone, więc pętla `for` nic nie wniesie.

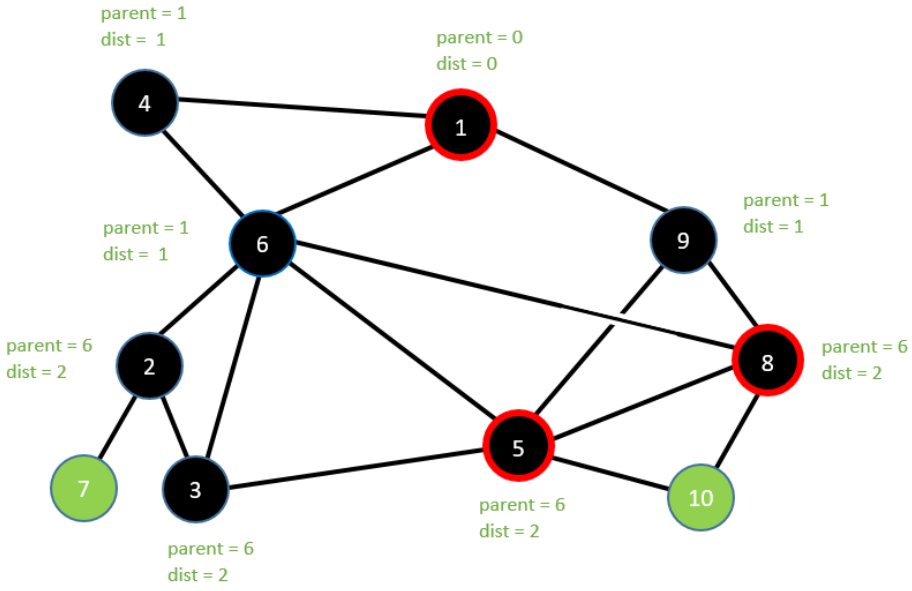
Następny w kolejce Q jest wierzchołek 6 – zdejmujemy go, podstawiamy za v i przeglądamy jego listę sąsiadów (pętlą `for`):



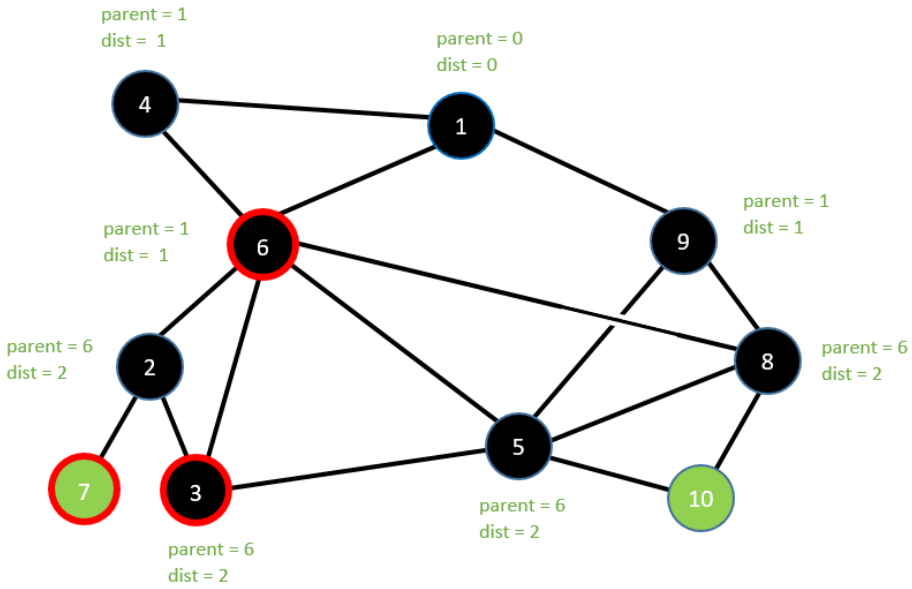
Lista sąsiedztwa zawiera wierzchołki 1, 4, 2, 3, 5, 8 (patrz poprzednia sekcja), z czego wierzchołki 1 oraz 4 są już odwiedzone, zatem do kolejki Q wędrują wierzchołki 2, 3, 5 oraz 8 (dla tych wierzchołków *parent*-em jest wierzchołek 6, a odległość wynosi 2):



Pierwszy w kolejce jest teraz wierzchołek 9 – zdejmujemy, podstawiamy za v i rozpatrujemy jego listę sąsiedztwa:

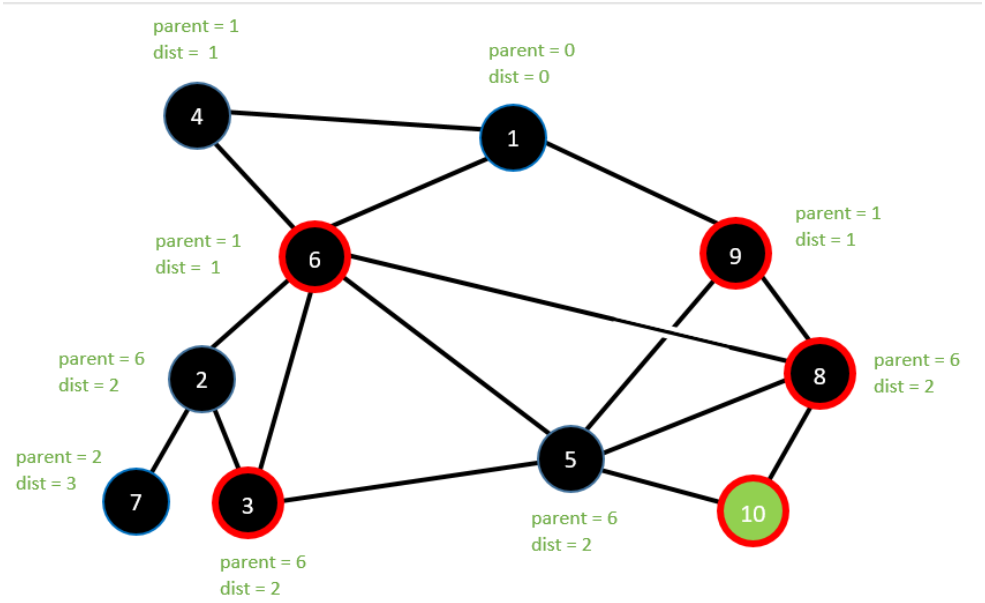


Nie ma tu nic nieodwiedzonego, więc pętal `for` dochodzi do końca bez żadnych ekscesów. Następny w kolejce Q jest wierzchołek 2:

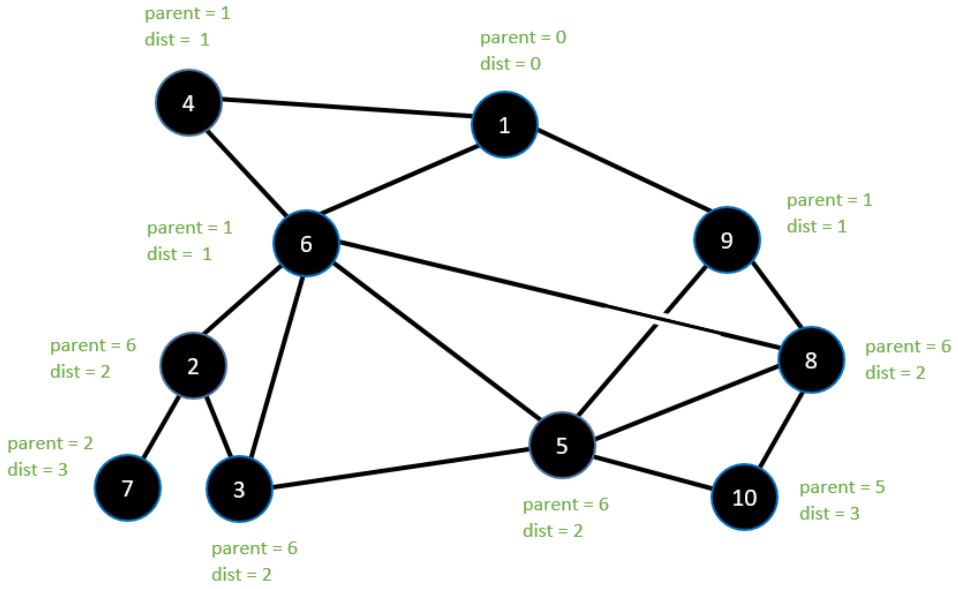


Jego lista sąsiedztwa to 6, 7, 3, z czego tylko 7 jest nieodwiedzony i ma szansę powalczyć. Otrzymuje 2 on jako *parent*-a i 3 jako odległość – no i wędruje do kolejki.

W kolejce czeka teraz wierzchołek 3 – nie ma nieodwiedzonych sąsiadów, więc nie będziemy się nad nim rozczulać. Następny proszę! Następny to wierzchołek 5:



Jedynym nieodwiedzonym sąsiadem jest wierzchołek 10 – otrzymuje on *parent*-a 5 i odległość 3:



Wierzchołek 10 wysyłamy do kolejki – rezydują tam jeszcze wierzchołki 8 i 7, ale żaden z lokatorów kolejki nie ma już nieodwiedzonych sąsiadów, więc pętla `while` obejdzie na jałowym biegu do końca i możemy wieszać wiechę!

Jeszcze dopiszemy funkcję `print_BFS)` oraz podrasujemy nieco funkcję `main()`, aby móc nacieszyć się wynikami przeszukania grafu:

```
void print_BFS(vector<vector<int>> &G, int V,
               vector<int> &parent, vector<int> &dist)
{
  for(int v = 1; v <= V; v++)
    cout << v << ": parent=" << parent[v] << " dist=" << dist[v] << endl;
}
int main()
{
  int V, E;
  vector<vector<int>> G;
  read_graph(G, V, E);
//  print_graph(G, V);
  vector<bool> visited(V + 1);
  vector<int> parent(V + 1), dist(V + 1);
  BFS(G, V, 1, visited, parent, dist);
  print_BFS(G, V, parent, dist);
}
```

Chyba nie trzeba rozwodzić się nad tym, dlaczego zadeklarowane tu wektory posiadają rozmiary $V + 1$. Warto pamiętać, że typ danych `vector<>` jest automatycznie inicjalizowany przy deklaracji – właśnie takimi wartościami, jakie są nam potrzebne.

Wywołanie funkcji `print_graph()` zostało *wykomentowane*.

Po uruchomieniu tego programu i wpisaniu danych naszego grafu dostajemy wyniki:

```
1: parent=0 dist=0
2: parent=6 dist=2
3: parent=6 dist=2
4: parent=1 dist=1
5: parent=6 dist=2
6: parent=1 dist=1
7: parent=2 dist=3
8: parent=6 dist=2
9: parent=1 dist=1
10: parent=5 dist=3
```

czyli dokładnie tak, jak było na ostatnim rysunku.

Kilka ważnych uwag

Nazwa algorytmu – *przeszukiwanie wszerz* – jest związana z faktem, że na każdym kroku (przy przetwarzaniu każdego wierzchołka) zajmujemy się tylko jego najbliższymi sąsiadami, tworząc swoistą *tyralierę* posuwającą się krok po kroku. Nietrudno zauważyć, że w ten sposób wyznacza się najkrótsze możliwe odległości pomiędzy wierzchołkiem, a wszystkimi innymi wierzchołkami grafu. A co z przebytą trasą? Czy da się ją odtworzyć? Odpowiedź jest twierdząca – wystarczy popatrzyć się na wektor *pattern*: zawiera on numery poprzedników w wedrówce do danego wierzchołka. Możemy bez problemu odtworzyć trasę od tyłu: weźmy na przykład wierzchołek 10. Jego poprzednikiem jest wierzchołek 5, poprzednikiem wierzchołka 5 jest 6, a poprzednikiem wierzchołka 6 jest wierzchołek startowy (1). Odwracając kolejność otrzymamy następującą ścieżkę:**

$$1 \to 6 \to 5 \to 10.$$

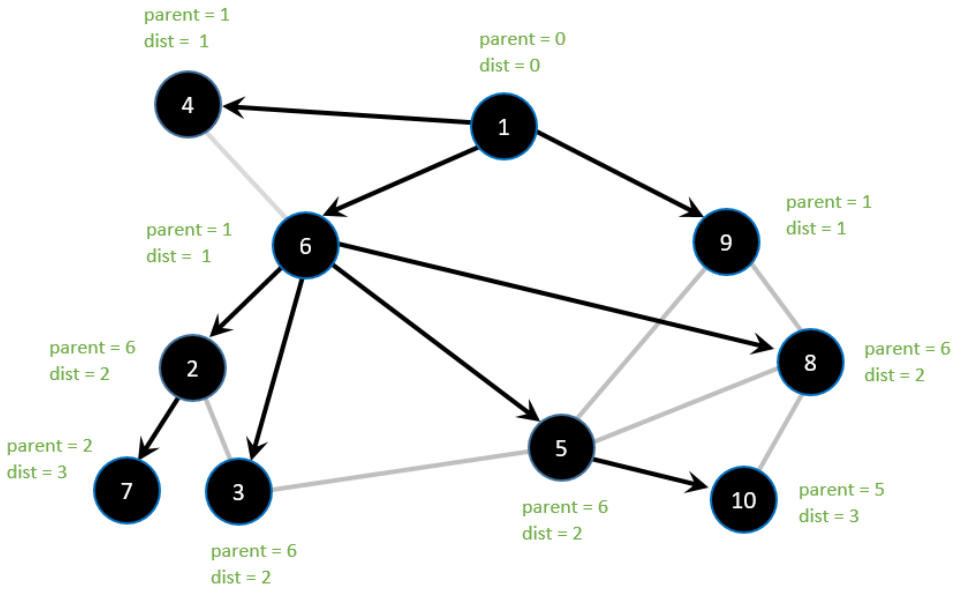
Ta ścieżka (najkrótsza) nie jest wyznaczona jednoznacznie, wszakże poniższa ścieżka ma taki sam początek, taki sam koniec i taką samą długość:

$$1 \to 9 \to 8 \to 10.$$

Jedno jest pewne: najkrótsza droga od wierzchołka 1 do wierzchołka 10 ma długość 3 – i tego się trzymajmy.

**Ścieżką w grafie nazywamy ciąg jego krawędzi: koniec danej krawędzi (wierzchołek) jest początkiem następnej.

Możemy narysować wszystkie ścieżki, które zostały znalezione w wyniku algorytmu BFS (po odwróceniu kolejności wierzchołków na każdej ścieżce):



Co dostaliśmy? Tak, tak, drzewo! I to ukorzenione w wierzchołku startowym 1.[††] I taki właśnie jest rezultat algorytmu BFS.

[††] Qrczę, ciągle jakieś drzewa nam tu wyrastają, chyba trzeba będzie poświęcić im oddzielne *Miaukoty*...