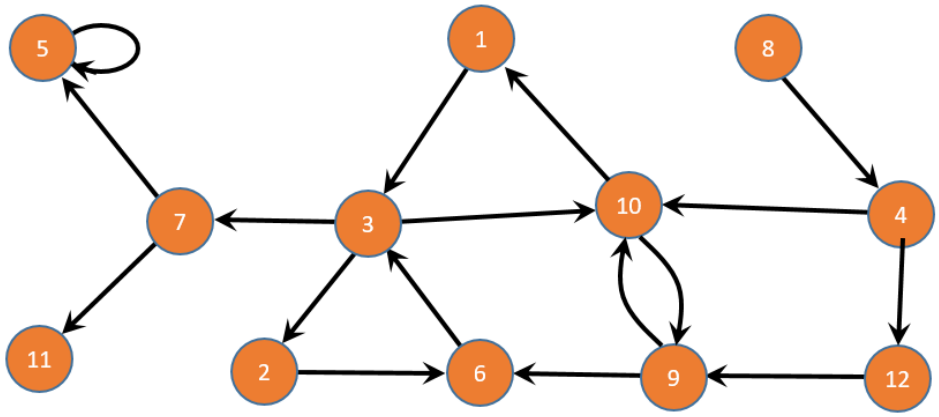
Z impetem w głąb!

#graf_skierowany
#czas_wejścia
#czas_wyjścia

Czas na nieco inny sposób przeszukiwania grafu: przeszukiwanie w głąb* (ang. **DFS**, **Depth-First Search**, wymawiaj: *depθ ferst sercz*). Ta forma spaceru po grafie wykorzystywana jest w niezliczonej ilości zaawansowanych algorytmów grafowych oraz zastosowaniach praktycznych. Zazwyczaj algorytm DFS stosujemy do grafów skierowanych (i takie przykłady będziemy prezentować), choć bywa, że przydaje się on w grafach nieskierowanych (choćby w równoważnej do BFS metodzie znajdowania spójnych składowych grafu.) W kolejnych podrozdziałach omówimy dwa ważne zastosowania algorytmu DFS – sortowanie topologiczne i wyznaczanie silnie spójnych składowych – natomiast tutaj przedstawimy podstawy tego algorytmu.

Weźmy sobie przykładowy graf skierowany (ang. **directed**, wymawiaj: *dajrekted*):



Jak widać, nieco poluzowaliśmy zasady, dopuszczając krawędzie zaczynające się i kończące w tym samym wierzchołku. Poza tym pomiędzy wierzchołkami 9 i 10 mamy 2 krawędzie, ale wystarczy się przyjrzeć: jedna wiedzie od 9 do 10, a druga od 10 do 9. Prócz tego trzeba zwrócić baczną uwagę na kolejność numerów wierzchołków w opisie poszczególnych krawędzi w danych wejściowych, na przykład mamy krawędź $7 \to 11$, ale nie mamy krawędzi $11 \to 7$. (W przypadku grafu nieskierowanego nie było takiego problemu.) Dane wejściowe dla powyższego grafu mogą wyglądać tak:

```
12 17
1 3
10 1
8 4
```

*Z obywatelskiego obowiązku zwracamy uwagę, że **wszerz** piszemy razem, natomiast **w głąb** – oddzielnie.

```
5 5
7 5
3 7
3 10
4 10
7 11
3 2
6 3
2 6
9 6
9 10
10 9
12 9
4 12
```

Reprezentacja grafu nie ulegnie zmianie: ilość wierzchołków V, ilość krawedzi E, wektor list sąsiedztwa G, ale musimy napisać nową wersję funkcji `read_graph()`:

```
void read_directed_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);
  }
}
```

Różnica jest chyba widoczna, prawda?

Funkcja `print_graph()` poda nam następujące listy sąsiedztwa:

```
1: 3
2: 6
3: 7 10 2
4: 10 12
5: 5
6: 3
7: 5 11
8: 4
9: 6 10
10: 1 9
11:
12: 9
```

Przeszukiwanie grafu skierowanego nie zawsze idzie bardzo gładko: nawet jeśli graf jest w jednym kawałku, to jednokierunkowość krawędzi może powodować kłopoty komunikacyjne.

Dlatego przeszukiwanie wykonuje się w taki sposób, jak w przypadku badania spójności grafu, czyli wykonując pętlę po wierzchołkach – jeśli napotkamy nieodwiedzony wierzchołek, to wtedy od niego puszczamy właściwe przeszukiwanie:

```
void DFS(vector<vector<int>> &G, int V,
         vector<bool> &visited, vector<int> &parent,
         int &t, vector<int> &start, vector<int> &finish)
{
  for(int v = 1; v <= V; v++)
    if(!visited[v])
      DFS_visit(G, V, v, visited, parent, t, start, finish);
}
```

No, coś się dzieje! Parametrów funkcji jak maku! Niektóre już znamy i lubimy, jak G, V, *visited* (wystarczy `bool`) czy *parent*. Ale co to jest t, *start* i $finish$? Otóż podczas spaceru po grafie będziemy notować czas wejścia do danego wierzchołka (*start*) i czas zakończenia jego przetwarzania ($finish$). Przyjmiemy, że każda operacja wejścia do wierzchołka i przejścia krawędzi zajmuje jednostkę czasu. Upływ czasu będzie rejestrować zmienna t.

Wizytę w wierzchołku realizować będzie funkcja `DFS_visit()`:

```
void DFS_visit(vector<vector<int>> &G, int V, int s
         vector<bool> &visited, vector<int> &parent,
         int &t, vector<int> &start, vector<int> &finish)
{
  . . .
}
```

Odwiedzanym wierzchołkiem jest wierzchołek s:

```
visited[s] = true;
start[s] = ++t;
. . .
finish[s] = ++t;
```

W miejsce trzech kropek wstawimy pętlę spacerującą po liście sąsiedztwa wierzchołka s. Jak zwykle interesują nas tylko wierzchołki nieodwiedzone (dla których poprzednikiem staje się wierzchołek s):

```
for(int x : G[s])
  if(!visited[x])
  {
    parent[x] = s;
    DFS_visit(G, V, x, visited, parent, t, start, finish);
  }
```

Jednym słowem, bardzo prosty i zwięzły algorytm, ale ja on tak naprawdę działa? Jeśli funkcja `DFS_visit()` złapie w swe szpony jakiś wierzchołek, wtedy wyszukuje pierwszego nieodwiedzonego sąsiada i bierze go w obroty, wywołując swojego potomka dla tegoż sąsiada. U sąsiada jest

podobnie, szukany jest jego najbliższy nieodwiedzony sąsiad, dla niego odpalany jest kolejny potomek funkcji – i tak aż do spodu, aż nie będzie co brać. Wtedy zgodnie z regułami rekurencji następuje zwijanie układu (w istocie: drzewa) potomków, począwszy od dołu. Na każdym poziomie uruchomiona jest pętla po sąsiadach wierzchołka, dla którego działa dany potomek funkcji. Oznacza to, że instrukcja nadająca wartość zmiennej $finish[s]$ zostanie wykonana dopiero po zwinięciu się całego drzewa potomków z rozgałęzieniami, czyli po zakończeniu przetwarzania wszelkich danych, jakie wyciągniemy z wierzchołków będących w polu rażenia tego drzewa.

Przećwiczymy działanie algorytmu DFS na naszym grafie wykorzystując poniższą funkcję `main()`:

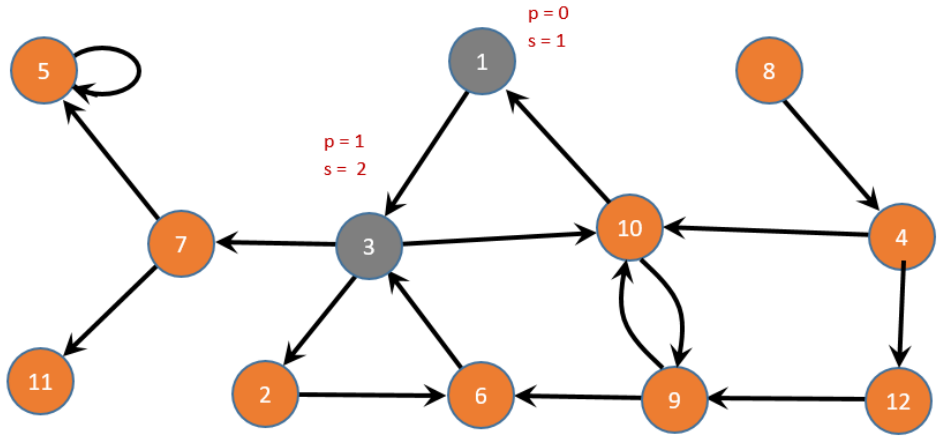
```
int main()
{
  int V, E;
  vector<vector<int>> G;
  read_directed_graph(G, V, E);
  print_graph(G, V);
  vector<bool> visited(V + 1);
  vector<int> parent(V + 1), start(V + 1), finish(V + 1);
  int t = 0;
  DFS(G, V, visited, parent, t, start, finish);
  print_DFS(G, V, parent, start, finish);
}
```

Jeszcze tylko napiszemy funkcję `print_DFS()`:

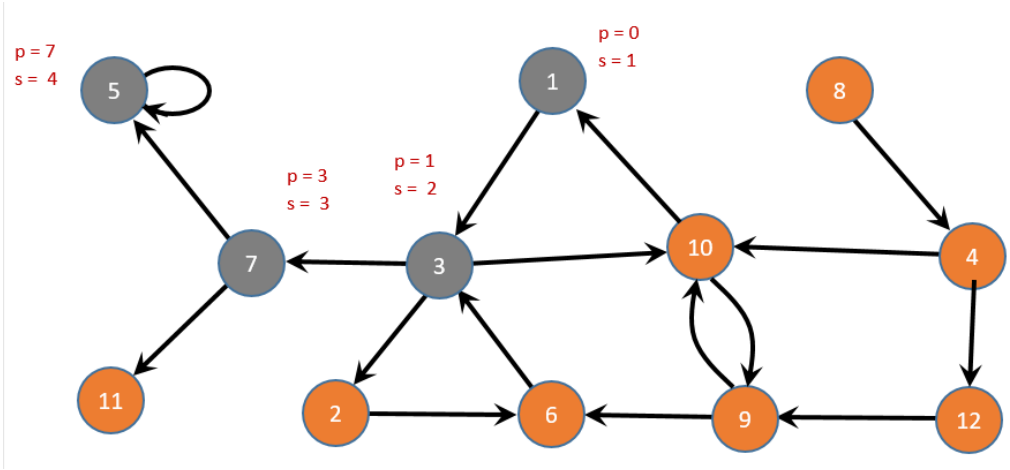
```
void print_DFS(vector<vector<int>> G, int V, vector<int> &parent,
               vector<int> &start, vector<int> &finish)
{
  for(int v = 1; v <= V; v++)
    cout << v << ": p=" << parent[v]
         << " s=" << start[v] << " f=" << finish[v] << endl;
}
```

... i już tuptamy po grafie. Prześledzimy działanie funkcji `DFS()` i kolejnych wywołań funkcji `DFS_visit()`. Odwiedzone wierzchołki będziemy znaczyć kolorem szarym, natomiast wierzchołki w pełni przetworzone – kolorem czarnym.

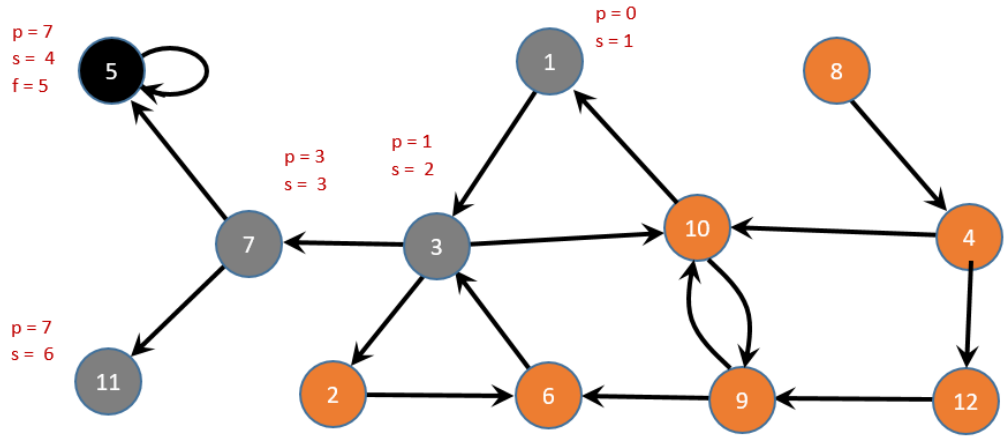
Pętla `for` w funkcji `DFS()` już przy pierwszym obiegu napotyka na nieodwiedzony wierzchołek ($v = 1$) i uruchamia funkcję `DFS_visit()` dla tego wierzchołka. Ta z kolei ma do rozpracowania (docelowo) jego listę sąsiedztwa, która liczy tylko jeden wierzchołek o numerze 3, który odwiedzamy (nie zapominając o ustawieniu odpowiednich wartości czasu *start* – odpowiednio 1 i 2):



Pierwszym sąsiadem wierzchołka 3 jest wierzchołek 7, z kolei pierwszym sąsiadem tego drugiego jest wierzchołek 5.[†] Odwiedzamy je po kolei i otrzymujemy:



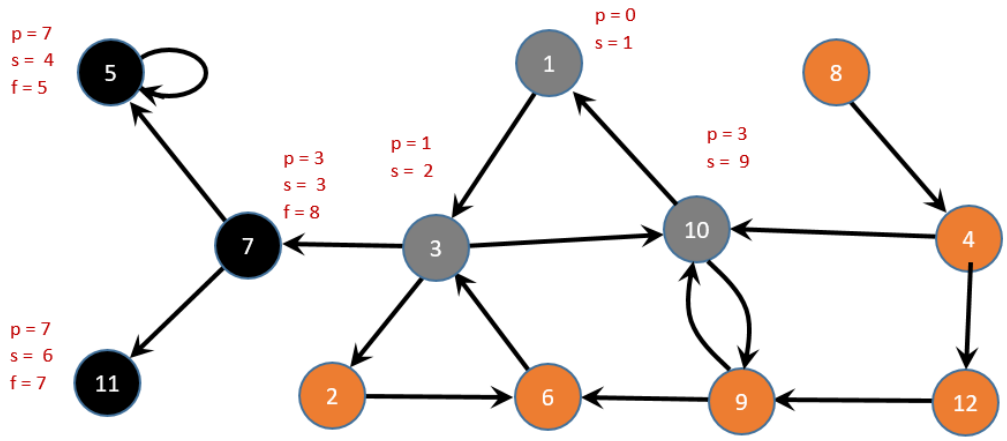
Jesteśmy przy wierzchołku 5 – z nim jest taka sprawa, że ma on sąsiada (samego siebie), ale jest on już odwiedzony, więc nic tu nie zwojujemy. Zatem ta instancja funkcji `DFS_visit()` kończy działanie, tym samym uznajemy wierzchołek 5 za całkowicie przetworzony (nadajemy odpowiednią wartość zmiennej $finish[5]$) i wycofujemy się do wierzchołka 7. Ma on jeszcze jednego nieodwiedzonego sąsiada – wierzchołek 11, który odwiedzamy:



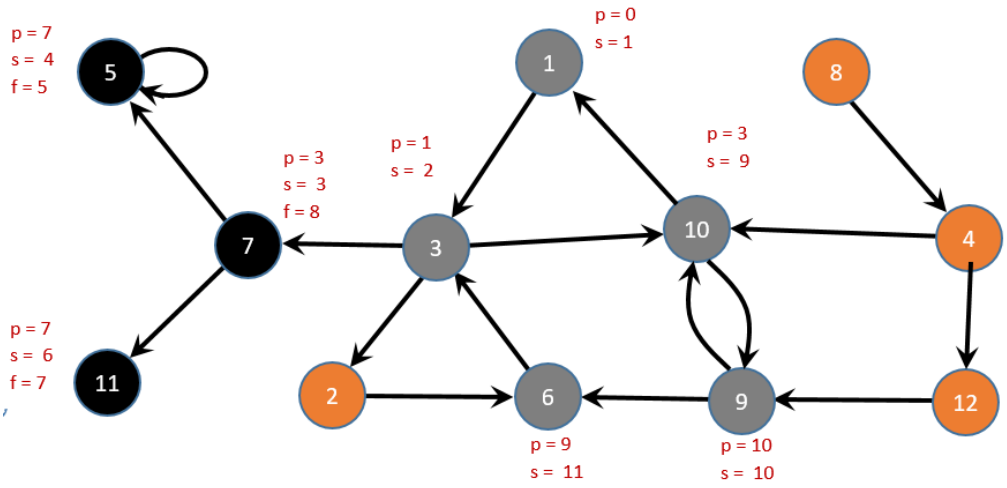
Ten wierzchołek nie ma sąsiadów, więc od razu uznajemy go za przetworzony i wycofujemy się rakiem do jego *parent*-a, czyli wierzchołka 7. Ten z kolei można uznać za przetworzony, wracamy

[†]Cały czas mamy na uwadze listy sąsiedztwa wypisane przez funkcję `print_graph()`.

więc do wierzchołka 3, gdzie jako drugi na jego liście sasiedztwa czai się wierzchołek 10:‡

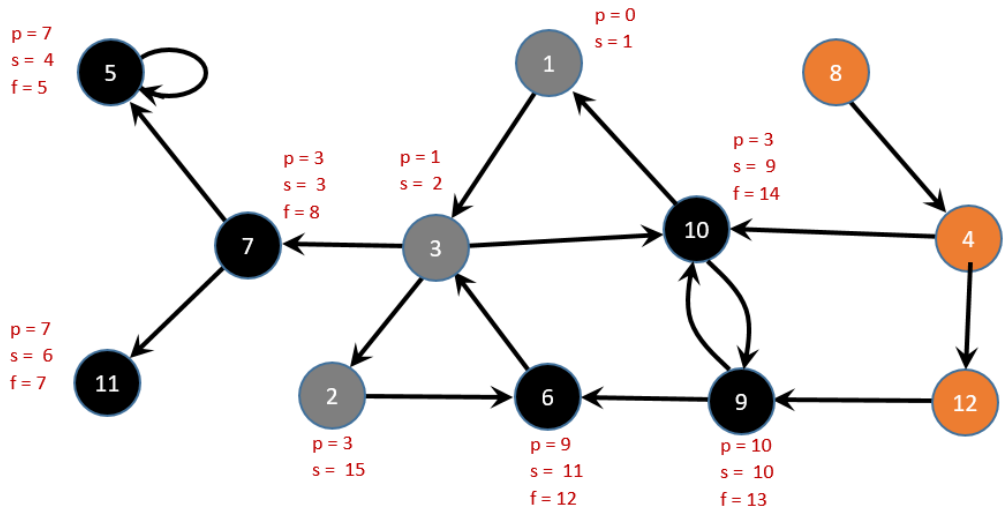


Wierzchołek 10 dobrze rokuje, bo ma nieodwiedzonego sąsiada 9, a ten z kolei ma nieodwiedzonego sąsiada 6 (pomijamy jałowych, odwiedzonych sąsiadów):

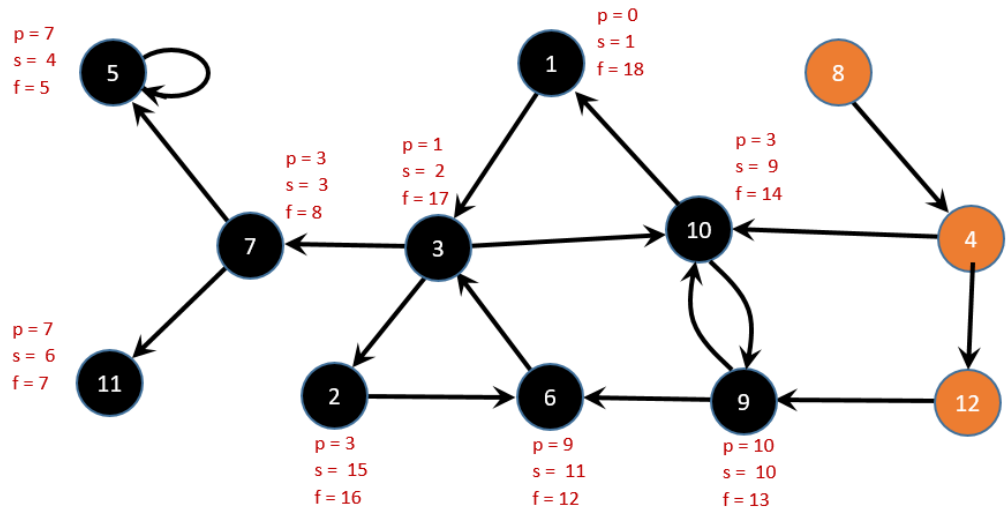


Mamy ślepy zaułek, więc wycofujemy się po śladach wstecz aż do wierzchołka 3, gdzie znajdziemy jeszcze jednego nieodwiedzonego sąsiada – wierzchołek 2. Wchodzimy do niego z czasem *start* o wartości 15:

‡Tak przy okazji zauważmy, że przez cały ten czas wierzchołek 1 czeka sobie cierpliwie na spodzie rekurencyjnego stosu wywołań funkcji `DFS_visit()`. Ale przyjdzie na niego czas...

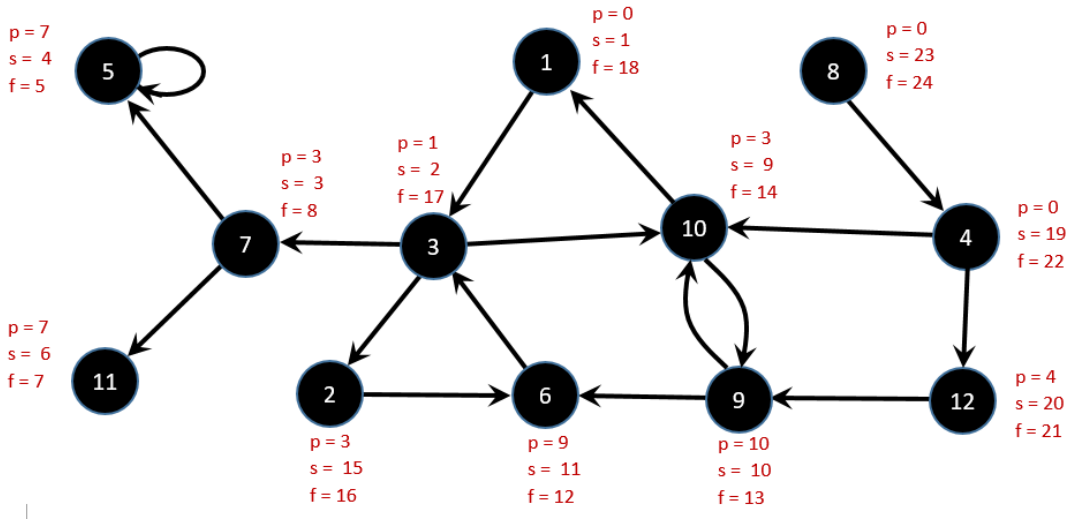


Długo tu nie zagościmy, bo nieodwiedzonych sąsiadów tu nie stwierdzono. Robimy odwrót do wierzchołka 3 (ruch tu jak na skrzyżowaniu pod DH „Jubilat”), aż w końcu wracamy do punktu wyjścia: wierzchołka 1 z czasem zakończenia 18:



Ocieramy pot z utrudzonego czoła i stwierdzamy ze zgrozą, że nie odwiedziliśmy jeszcze kilku wierzchołków! A jak się obrażą?!

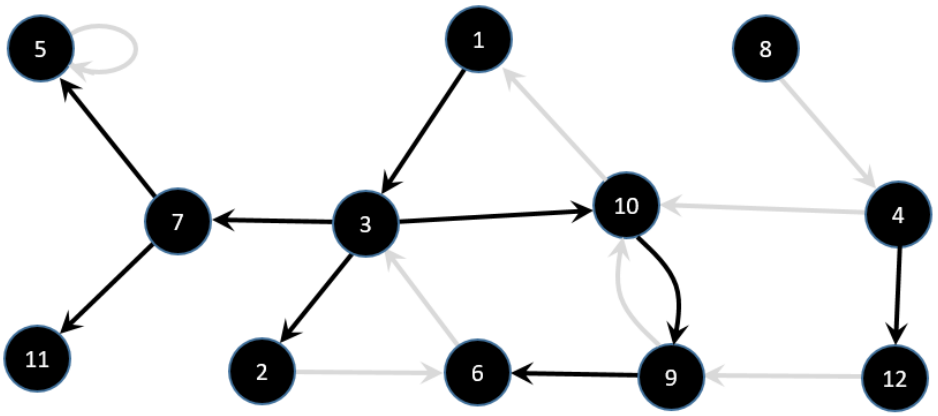
Nie bój żaby, mamy w odwodzie jeszcze nadrzedną funkcję, czyli `DFS()`. Wracając do wierzchołka 1 pozamykaliśmy wszystkie instancje funkcji `DFS_visit()`, ale pętla `for` po wierzchołkach grafu (zmienna v) działa dalej. Kolejne wartości v to 2 i 3 odpowiadające odwiedzonym i w pełni przetworzonym wierzchołkom, więc nic tu po nas. Dopiero $v = 4$ otworzy nam dostęp do *terra incognita* i pozwoli dotrzeć do wierzchołka 12. Na koniec $v = 8$ odsłoni pojedynczy wierzchołek o tym właśnie numerze:



Funkcja `print_DFS()` wyprodukuje następujący wydruk:

```
1: p=0 s=1 f=18
2: p=3 s=15 f=16
3: p=1 s=2 f=17
4: p=0 s=19 f=22
5: p=7 s=4 f=5
6: p=9 s=11 f=12
7: p=3 s=3 f=8
8: p=0 s=23 f=24
9: p=10 s=10 f=13
10: p=3 s=9 f=14
11: p=7 s=6 f=7
12: p=4 s=20 f=21
```

Jeśli pozostawimy w naszym grafie tylko krawędzie wyróżnione w wektorze *pattern*[§], wtedy otrzymamy zbiór drzew, czyli *las*:



W następnych podrozdziałach nauczymy się wykorzystywać wiedzę o strukturze grafu zapisaną w wektorach związanych z upływem „czasu” (na razie tylko w wektorze $finish$).

[§]Analogicznie jak w przypadku algorytmu BFS należy odwrócić zależności.