Grafy

Karol Janik

January 2021

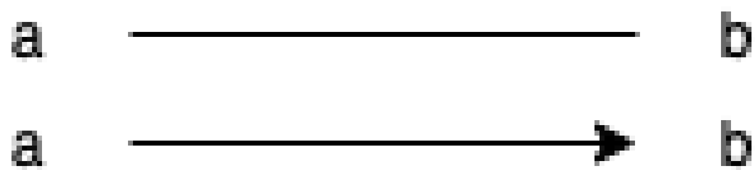
Każde zadanie ma swój rytm. Jak w muzyce. Jeśli się dobrze wsłuchać, można zobaczyć problem z góry i zauważyć wszystkie zasadzki i niebezpieczeństwa. - Ukochane Równanie Profesora, Yōko Ogawa

Spis treści

1 Wprowadzenie

1.1 Definicja grafu

Graf to system, który zapisujemy $G = (V, E)$, gdzie V oznacza zbiór skończony, którego elementy nazywamy **wierzchołkami**, a E - zbiór krawędzi czli par wierzchołków ze zbiorku V, przy czym, dokładniej E jest podzbiorem zbioru par uporządkowanych $\{(a, b) : a, b \in V \wedge a \neq b\}$ i wtedy graf jest **zorientowany**, a krawędzie są oznaczone strzałkami łączącymi wierzchołki, albo E jest podzbiorem zbioru wszystkich dwuelementowych podzbiorów zbioru V i wtedy graf jest **niezorientowany**, a krawędzie są oznaczone liniami.[1]



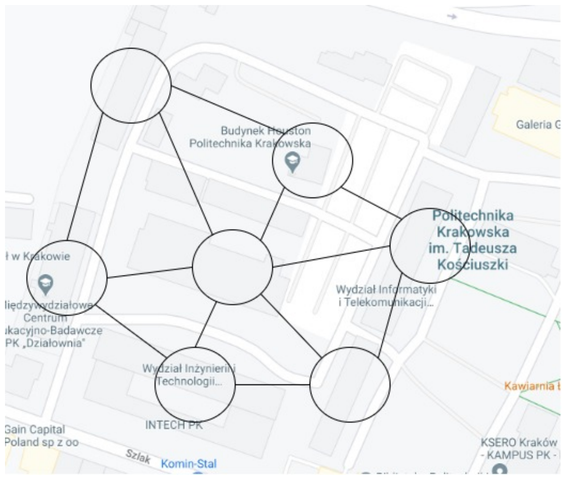
Rysunek 1: Dwa rodzaje krawędzi w grafach

1.2 Terminologia od środka

Przyjrzyjmy się nieco terminologii dotyczącej. Elementy na grafie nazywane są wierzchołkami. Wierzchołki reprezentują obiekty znajdujące się na grafie, a mianowicie rzeczy, które połączone są jakąś relacją, tak jak ludzie, czy np. miasta. Połączenie między dwoma wierzhołkami nazywa się krawędzią, krawędź reprezentuję relację, która łączy wierzchołki.

Przykłady grafów w codziennym życiu:

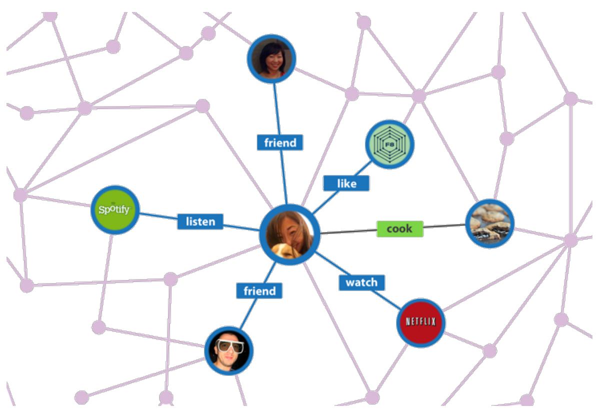
- Najbardziej lokalnym przykładem możebyć kampus Politechnikii Krakowskiej, budynki mogą reprezentować wierzchołki, a ścieżki łączące te budynki byłyby krawędziami.



Źródło: Google maps

Rysunek 2: Kampus PK jako graf

- Facebook jest oparty na grafach, każda osoba jest wierzchołkiem, a krawędzie reprezentują działania, które wykonujemy na naszym profilu.

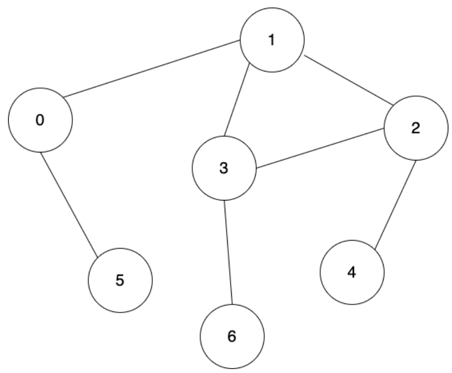


Źródło: Google grafika

Rysunek 3: Facebook graf

1.3 Graf zorientowany vs niezorientowany

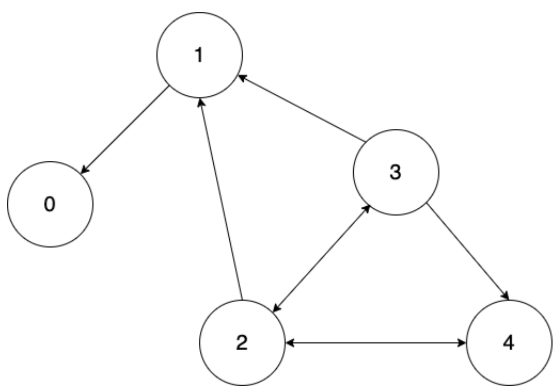
Jeżeli wszystkie krawędzię na grafie pokazują zależności między dwoma wierzchołkami działającymi w dowolnym kierunku, nazywa sie go grafem nieskierowanym. Obraz nieskierowanego grafu wygląda następująco:



Źródło: Własne opracowanie

Rysunek 4: Niezorientowany graf

Niestety, nie wszystkie krawędzie na grafie są takie same. Czasami relacje między dwoma wierzchołkami idą tylko w jednym kierunku. Taka zależność nazywana jest grafem skierowanym. Przykładem może być mapa miasta, której niektóre ulice są jednokierunkowe. W rzeczywistości nawet dwukierunkowa ulica powinna być reprezentowana jako dwie krawędzie, jedna prowadząca z lokalizacji A do B, druga z B do A. Wizualizacja grafu skierowanego:



Źródło: Własne opracowanie

Rysunek 5: Zorientowany graf

Mówiąc bardziej formalnie, krawędź od wierzchołka $V1$ do $V0$ jest zbiorem $(V1, V0)$. Na grafie skierowanym zbiór ten jest uporządkowany, więc nawet, jeżeli istnieje krawędź $(V1, V0)$, to $(V0, V1)$ istnieć nie może. Inaczej jest w przypadku grafu nieskierowanego, tam zbiór wierzchołków jest nieuporządkowany, więc krawędz $(V1, V0)$ jest tą samą krawędzią, co $(V0, V1)$.

1.4 Jeszcze kilka definicji

Oto kilka innych terminów, które mogą się przydać przy omawianiu grafów.

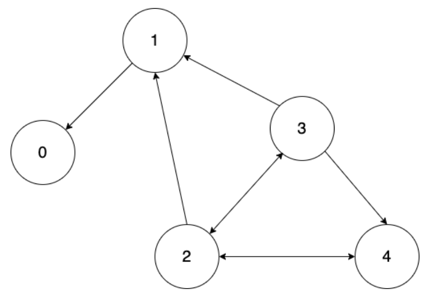
Określenie	**Definicja**
Krawędź	Pojedyńcze połączenie pomiędzy dwoma wierzchołkami
Sąsiadujący	Dwa wierzchołki sąsiadują ze sobą, jeżeli łączy je krawędź
Połączony	Graf jest połączony, jeżeli każdy wierzchołek ma ścieżkę do wszystkich innych wierzchołków
Sąsiad	Dwa wierzchołki są sąsiadami, jeżeli łączy je krawędź
Zbiór sąsiadów	Zbiór wszystkich wezłów które sąsiadują ze sobą
Scieżka	Sekwencja krawędzi, po której można podążać od jednego do drugiego wierzchołka
Cykl	Specjalny rodzaj ścieżki, która konczy się w tym samym wierzchołku, w którym się zaczęła
Ważony	Dany graf nazywamy ważonym, jeżeli zawiera krawędzie, którym przypisano pewną wagę. Wagą może być dowolna wielkość, np.: czas dojazdu, odległość, koszt trasy.

Tablica 1: Terminy dotyczące grafów

2 Graf jako struktura danych

Zdobyliśmy podstawową wiedzę dotyczącą grafów, wiemy, jak zwizualizować graf oraz znamy podstawowe pojęcia. Przyszedł czas na reprezentacje grafu w komputerze.

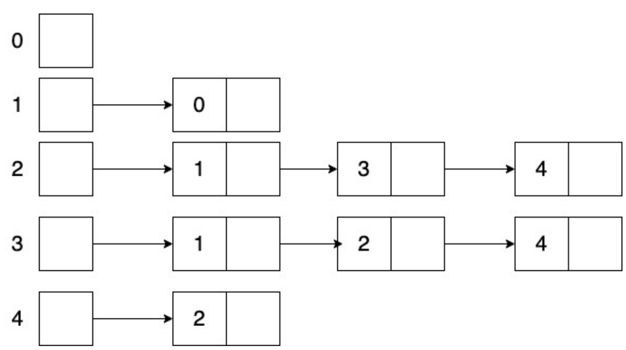
Wyobraźmy sobie, że chemy zbudować graf, podobny do tego:



Oto podstawowe implementacje grafu.

2.1 Listy sąsiedztwa

Jedną ze struktur danych, którą możemy wykorzystać do implementacji grafu, jest lista sąsiedztwa. W takiej strukturze danych tworzona jest tablica o takim samym rozmiarze, jak liczba wierzchołków na wykresie. Każda pozycja w tablicy reprezentuje jeden z wierzchołków. Następnie każda lokalizacja w tablicy wskazuje na listę zawierająca indeksy do innych wierzchołków, które ze sobą sąsiadują.



Źródło: Własne opracowanie

Rysunek 6: Wizualizacja listy sąsiedztwa

W tej implementacji potrzebna jest pamięć rozmiaru $O(n + m)$

2.2 Macierz sąsiedztwa

Inną strukturą danych, której moglibyśmy użyć do przedstawienia krawędzi na wykresie, jest macierz sąsiedztwa. W tej strukturze danych mamy tablice, w której każdy element reprezentuje wierzchołek na wykresie. Jednak zamiast tablicy wskazującej na połączoną listę, wskazuje inną tablicę, reprezentującą możliwych sąsiadów. Macierz zawiera tylko wartości logiczne, prawdziwe, gdy między dwoma podanymi wierzchołkami jest krawędź, fałszywe, gdy krawędź nie istnieje. Każdy wierzchołek ma wiersz i kolumnę w macierzy, a wartości w macierzy mowią, czy istnieje krawędź.

	0	1	2	3	4
0	F	F	F	F	F
1	T	F	F	F	F
2	F	T	F	T	T
3	F	T	T	F	T
4	F	F	T	T	F

Źródło: Własne opracowanie

Rysunek 7: Wizualizacja macierzy sąsiedztwa

W tej implementacji jest potrzebna pamięć rozmariu $O(n^2)$

3 Grafy w programowaniu obiektowym

Rozdział ten zostanie poświęcony implementacji grafów za pomocą paradygmatnu, jakim jest programowanie obiektowe. W tym celu posłużę się językiem Python.

3.1 Reprezentacja wierzchołka

```
class Vertex:
    def __init__(self, node):
        self.id = node
        self.adjacent = {}

    def __repr__(self):
        return str(self.id) + ' adjacent ' + str([x.id for x in
                                            self.adjacent])

    def add_neighbor(self, neighbor, weight=0):
        self.adjacent[neighbor] = weight

    def get_connections(self):
        return self.adjacent.keys()

    def get_id(self):
        return self.id

    def get_weight(self, neighbor):
        return self.adjacent[neighbor]
```

Klasa **Vertex** (wierzchołek) używa słownika do śledzenia krawędzi, z którymi jest połączona oraz wagi każdej krawędzi. W tej klasie zaimplementowane są następujące metody:

- __init__ - metoda konstruktora, która inicjuje ID węzła oraz słownik, który zawierać będzie sąsiednie krawędzie
- __repr__ - metoda, która będzie wyswietlała nasz wierzchołek w terminalu jako ciąg postaci ID wierzchołka adjencet lista sąsiadów
- add_neighbor - metoda służąca do dodania wierzchołka sąsiadującego
- get_connections - metoda zwracająca wszystkich sąsiadów naszego wierzchołka - czyli klucze słownika **adjacent**
- get_id - metoda zwracająca identyfikator wierzchołka
- get_weight - metoda zwracająca 'wage' sąsiadującego wierzchołka

3.2 Reprezentacja grafu

```
class Graph:
    def __init__(self):
        self.vert_dict = {}
        self.num_vertices = 0

    def __iter__(self):
        return iter(self.vert_dict.values())

    def add_vertex(self, node):
        self.num_vertices += 1
        new_vertex = Vertex(node)
        self.vert_dict[node] = new_vertex
        return new_vertex

    def get_vertex(self, n):
        try:
            return self.vert_dict[n]
        except:
            raise ValueError(f'Vertex  {n} does not exist')

    def add_edge(self, frm, to, cost=0):
        if frm not in self.vert_dict:
            self.add_vertex(frm)
        if to not in self.vert_dict:
            self.add_vertex(to)

        self.vert_dict[frm].add_neighbor(self.vert_dict[to], cost)
        self.vert_dict[to].add_neighbor(self.vert_dict[frm], cost)

    def get_vertices(self):
        return self.vert_dict.keys()
```

Klasa **Graph** zawiera słownik **vert_dict**, który odzworowuje nazwy wierzchołków na obiekty klasy **Vertex**, której obiekty możemy wyświetlać w terminalu za pomocą przeciążonej metody **__repr__** dla klasy **Vertex**. Metody klasy **Graph**:

- __init__ - metoda konstruktora, która inicjuje słownik **vert_dict** oraz zmienna **num_vertices**, która będzie zawierała liczbę wierzchołków w grafie
- __iter__ - metoda ułatwiająca iteracje po wszystkich wierzchołkach na obiekcie
- get_vertex - metoda zwracająca konkretny wierzchołek w grafie, w przypadku niepowodzenia zgłasza błąd
- add_edge - metoda, która dodaje wierzchołek do grafu
- get_vertices - metoda, która zwraca wierzchołki w naszym grafie - czyli klucze słownika **vert_dict**

4 Grafy w Haskelu

4.1 Tworzenie grafu w haskellu

Haskell dostarcza nam implementacje grafów za pomocą biblioteki Data.Graph - węzły w tej implementacji są zawsze identyfikowane przez liczbę całkowitą, a krawędzie są skierowane(krawędz od a do b nie oznacza krawędzi od b do a) oraz krawędzie nie posiadają wag. Isnieją dwa sposoby tworzenia grafu:

- Używamy funkcji *graphFromEdges*, która tworzy graf na podstawie listy sąsiedztwa; każdy z nich jest identyfikowany za pomocą klucza i przechowuje wartość dla każdego wierzchołka, a także listę sąsiadów - czyli listę dowolnych innych węzłów, które są z nim połączonych. Funkcja *graphFromEdges* pobiera listę trójek (value, key, [key]) - ostatnim elemetem jest wspomniana lista sąsiadów. Funkcja zwraca nam graf, ale także dwie funkcje. Pierwsza funkcja typu Vertex → (node, key, [key]) mapuje identyfikator wierzchołka z grafu na odpowiednią informacjęo nim, podczas gdy druga, typu Key → Maybe Vertex implementuje odwrotne odwzorowanie, czyli mapuje od kluczy do identyfikatorów wierzchołka[2]
- *buildG* - funkcja która przyjmuje jako parametry krotkę z minimalnymi i maksymalnymi identyfikatorami wierzchołków oraz listę krotek odpowiadających każdej skierowanej krawędzi na wykresie.[2]

4.2 Implementacja za pomocą funkcji buildG

```haskell
import Data.Graph

myGraph :: Graph
myGraph = buildG (103,2013)
             [(1302,1614),(1614,1302),(1302,2013),(2013,1302),(1614,2013)
             ,(2013,1408),(1408,1993),(1408,917),(1993,917),(917,103),(103,917)]
main = do
    print $ "The edges are " ++ (show.edges) myGraph
    print $ "The vertices are " ++ (show.vertices) myGraph
```

Jak działa powyższy kod?

- Importujemy biblioteke Data.Graph
- konstruujemy graf za pomocą funkcji buildG dostarczonej przez powyższą biblioteke
- Wypisujemy w terminalu graf - czyli jego wierzchołki i krawędzie.

4.3 Własna implementacja grafu

```
data Vertex = Vertex {
                        vertexLabel :: [Char]
                      , vertexNeighbors :: [[Char]]
                    } deriving Show

data Graph = Graph [Vertex] deriving Show

printGraph :: Graph -> IO ()
printGraph (Graph []) = putStrLn ""
printGraph (Graph (x:y)) = do
    print x
    printGraph (Graph y)
    return ()

main :: IO ()
main = do
    let myGraph = Graph [
                            Vertex "L"  ["S1", "S2"      ]
                        ,   Vertex "S1" ["S5" ]
                        ,   Vertex "S2" ["S3", "S4"]
                        ,   Vertex "S3" ["S5"]
                        ,   Vertex "S4" ["S5"]
                        ,   Vertex "S5" ["H"]
                        ]
    printGraph $ myGraph
    return()
```

Jak działa kod?

1. Za pomocą konstruktora **Data** tworzymy swój typ **Vertex**, który zawiera w sobie identyfikator węzła oraz listę sąsiadujących z nim węzłów
    - Ostatni wers jest nazywany **deriving clause**, czyli klauzulą pochodną, oznacza to, że chcemy, aby kompilator automatycznie generował wystąpienie klasy Show dla naszego typu
2. następnie definiujemy **Graph** - czyli nasz typ grafu, który składa się z listy węzłów
3. Tworzymy funkcje do wyświetlania grafu
4. Wyświetlamy graf

Program w terminalu wyświetla nasz graf:

```
Vertex {vertexLabel = "L", vertexNeighbors = ["S1","S2"]}
Vertex {vertexLabel = "S1", vertexNeighbors = ["S5"]}
Vertex {vertexLabel = "S2", vertexNeighbors = ["S3","S4"]}
Vertex {vertexLabel = "S3", vertexNeighbors = ["S5"]}
Vertex {vertexLabel = "S4", vertexNeighbors = ["S5"]}
Vertex {vertexLabel = "S5", vertexNeighbors = ["H"]}
```

5 Algorytmy grafowe

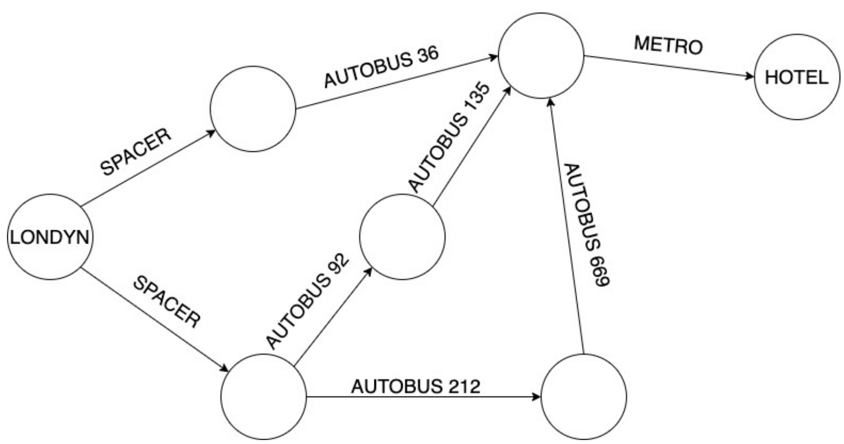
Rozdział ten skupia się na podstawowych **algorytmach** służacych do wykazania niektórych cech grafu. **Algorytmy** to skończony zestaw czynności koniecznych do wykonania pewnego zadania lub rozwiązania niektórych problemów. Zadaniem algorytmu jest zwrócenie poprawnych danych wyjściowych dla odpowiednich danych wejściowych.

5.1 Przeszukiwanie wszerz

Przeszukiwanie grafu polega na odwiedzeniu wszystkich wierzchołków, w celu zebrania informacji o grafie.

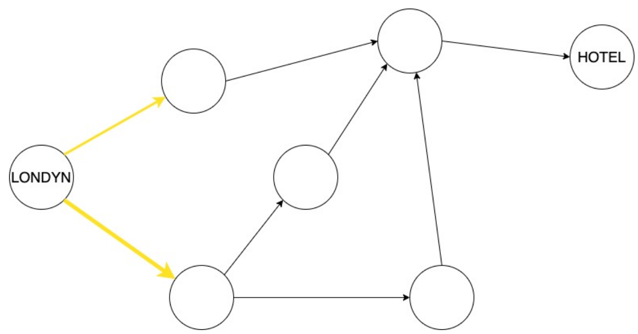
5.2 Zasada działania algroytmu

Wyobraźmy sobie, żę znajdujemy się w centrum Londynu i chcemy dotrzeć do hotelu, aby odpocząć. Zamierzamy korzystać z miejskiej komunikacji i chcemy ograniczyć liczbę przesiadek do minimum. Oto, jakie mamy możliwości:

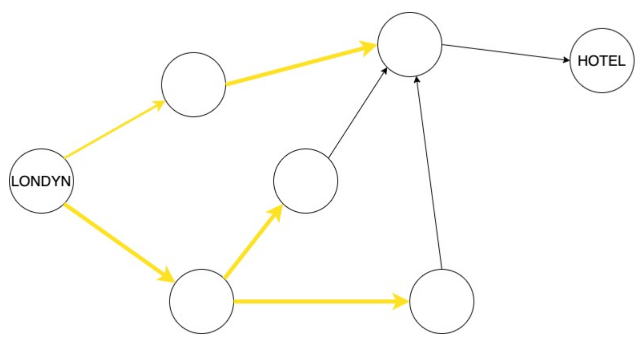


Źródło: Własne opracowanie, inspiracja [3]

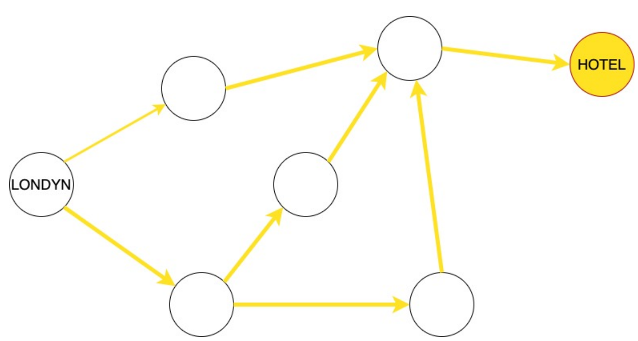
Oto wszystkie miejsca, do których można dotrzeć w pierwszym etapie:



Hotel nie został zaznaczony, co oznacza, że nie uda sie do niego dotrzeć w jednym kroku. Sprawdzamy, czy uda się dotrzeć w dwóch krokach:



Niestety, dalej nie dotarliśmy w dwóch krokach również nie dotarliśmy do naszego hotelu. Sprawdzamy, czy uda nam sie to zrobić w trzech krokach:



Udało się nam! Dotarliśmy do naszego hotelu, teraz możemy spokojnie odpoczywać po cieżkim dniu w centrum Londynu. Jak widać, droga z centrum do hotelu jest trzyetapowa. Istnieją również inne drogi, z których moglibyśmy skorzystać, ale każda z nich jest dłuższa (czteroetapowa). Tego rodzaju problem nazywa się **poszukiwaniem najkrótszej ścieżki**.

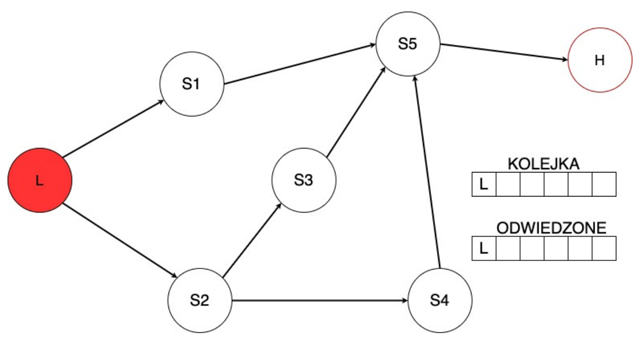
5.2.1 Algorytm przeszukiwania wszerz

Powyższy przykład zilustrował działanie algorytmu, teraz czas na opisanie jego zasady słowami. Procedura działa w następujący sposób

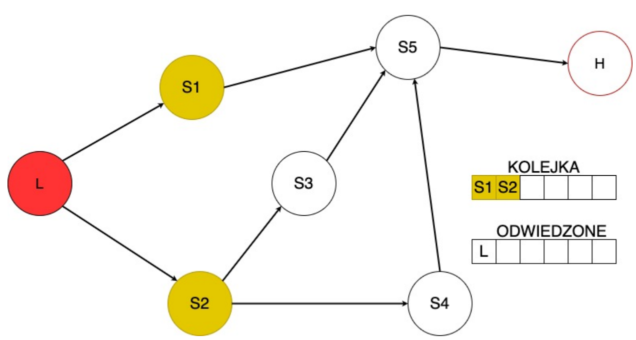
1. Dodajemy wierzchołek, z którego startujemy do pustej kolejki oraz zaznaczamy ten wierzchołek, jako odwiedzony.
2. Wyciągamy wierz z kolejki i dodajemy do niej sąsiadów, jeżeli nie są oznaczone jako odwiedzone.
3. Powtarzamy krok 2 aż kolejka będzie pusta.

Krok 1 jest inicjalizacją naszego algorytmu. Dodajemy startowy wierzchołek do kolejki i tym samym oznaczamy go jako odwiedzony. **Krok 2** jest głównym procesem wyszukiwania wszerz. Proces ten powtarza się, aby dodać wszystkich sąsiadów wyodrębnionego wierzchołka do kolejki. Wierzchołki dodane do kolejki są oznaczone jako nieodwiedzone.**Krok 3** jest warunkiem zakończenia pętli, gdy kolejka jest pusta - odwiedziliśmy wszystkie osiągalne wierzchołki z wierzchołka startowego, od którego zaczynaliśmy wyszukiwanie.

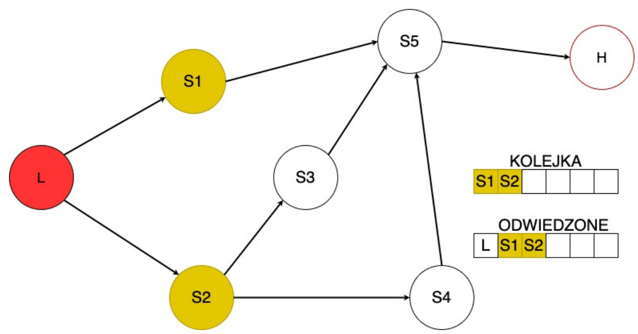
Sprawdźmy w jaki sposób ten algorytm działa, graf poniżej przedstawia początkowy stan algorytmu, w którym szukanie zaczynamy od wierzchołka **L**



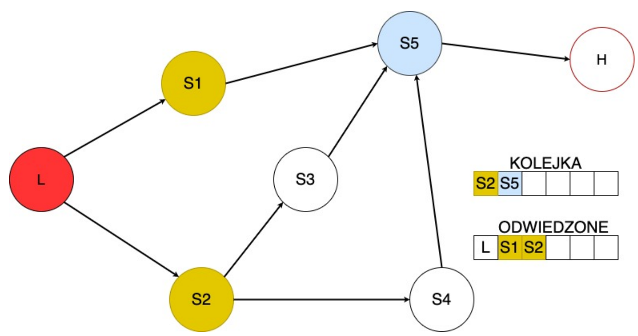
Pierwsze wyciągamy wierzchołek **L** z kolejki, sąsiedzi wiezchołka **L**, **S1** i **S2** nie zostali jeszcze odwiedzeni, więc dodajemy ich do kolejki:



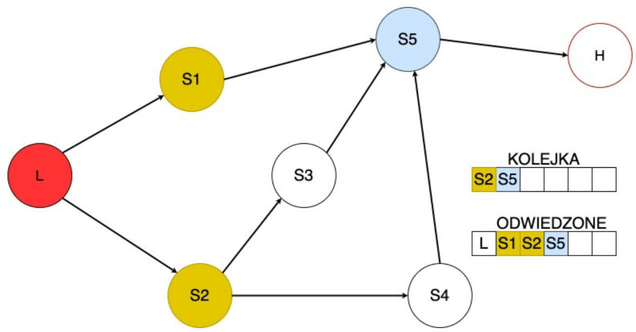
Zaznaczamy wierzchołki **S1** i **S2** jako odwiedzone, ponieważ zostały dodane do kolejki.



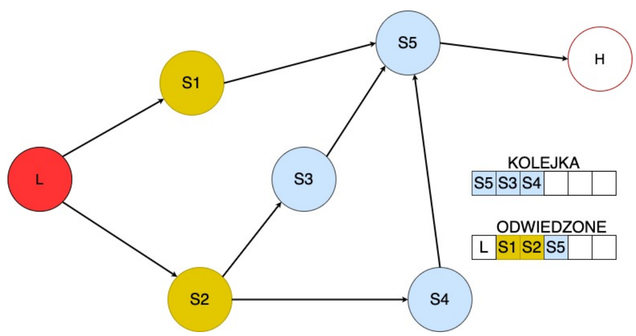
Następnię, sciągamy z kolejki **S1** i próbujemy dodać jego sąsiadów, czyli wierzchołek **S5**. Ponieważ graf jest skierowany - nie istnieje krawędz prowadząca z wierzchołka **S1** do **L**. Wyobraźmy sobię, że graf ten nie jest skierowany, czyl istnieje takie połączenie. Co w takim przypadku? Wierzchołek **S1** będzie miał dwóch sąsiadów: **L** oraz **S1**. Ponieważ wierzchołek **L** był wierzchołkiem startowym i jest zaznaczony, jako odwiedzony - nie dodajemy go do kolejki. W przypadku, gdy zostałby on dodany do kolejki, wpadlibyśmy w nieskończoną pętle i program nigdy nie zakończył by swojego działania. Innymi słowy, warunek w **kroku 3** był by zawsze prawdą, ponieważ kolejka nigdy nie byłaby pusta.



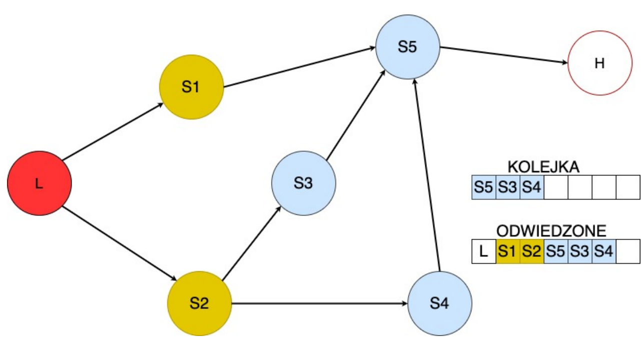
Dodajemy wierzchołek S5, jako odwiedzony ponieważ jest w kolejce.



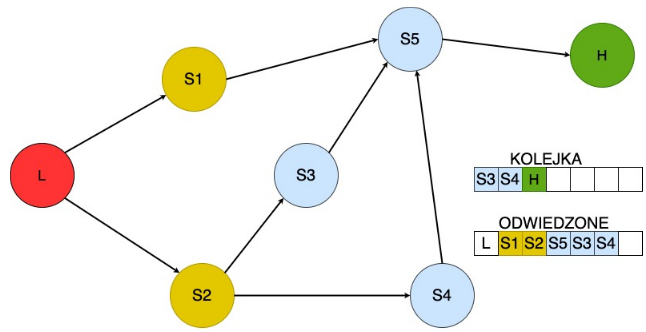
Kolejnym krokiem będzie ściągnięcie wierzchołka **S2** z kolejki, sprawdzenie jego sąsiadów i dodanie ich do kolejki.



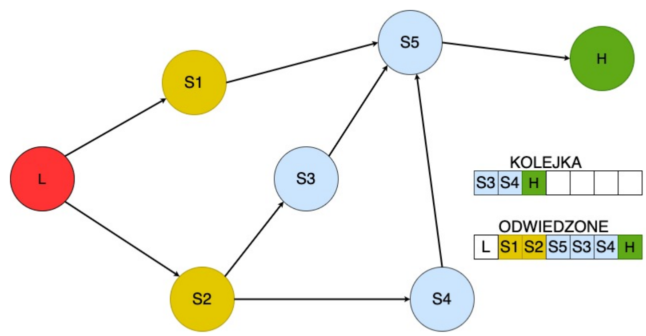
Następnie oznaczamy wierzchołki **S3**, **S4** jako odwiedzone, ponieważ są w kolejce.



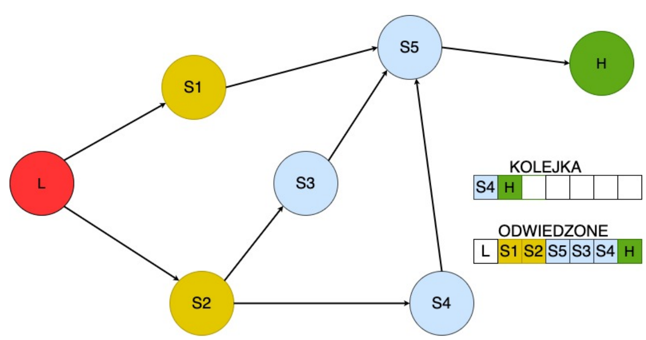
Kolejnym korkiem będzie wyciągnięcie wierzchołka **S5** z kolejki i dodanie jego sąsiadów - czyli wierzchołka **H**



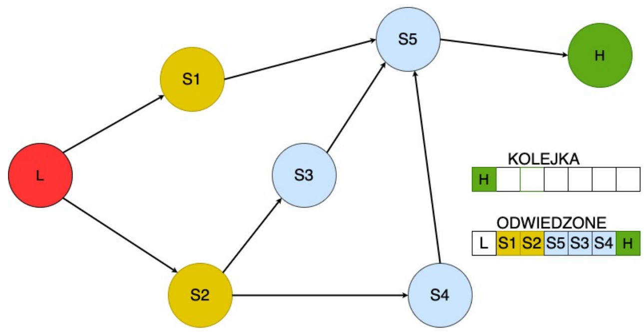
Ponieważ H znalazł się w kolejce - oznaczamy go jako odwiedzony.



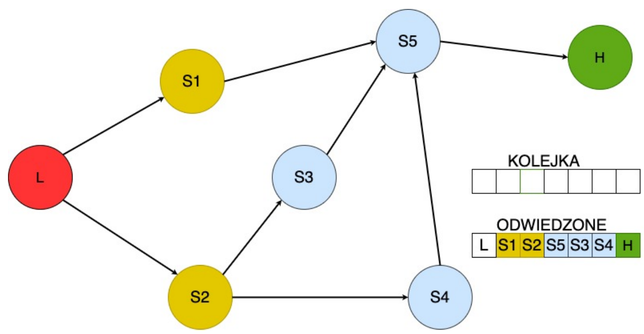
Następnię ściągamy z kolejki wierzchołek **S3**, sprawdzamy jego sąsiadów - ponieważ jedynym jego sąsiadem jest wierzchołek **S5**, który został już odwiedzony - nie dodajemy go do kolejki ponownie.



Sytuacja przy ściąganiu z kolejki wierzchołka **S4** jest taka sama - jedynym jego sąsiadem jest **S5**, który już został przez nas odwiedzony, więc nie dodajemy go do kolejki.



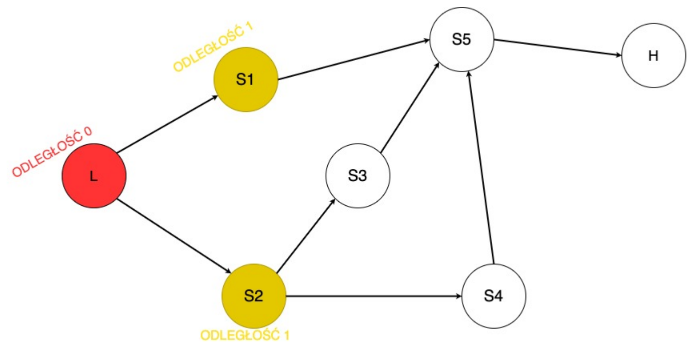
Następnie, ściągamy wierzchołek **H**, nie ma on żadnych sąsiadów, więc nie dodajemy nic do naszej kolejki - lista jest pusta, **krok 3** nie jest prawdą, więc algorytm kończy działanie.



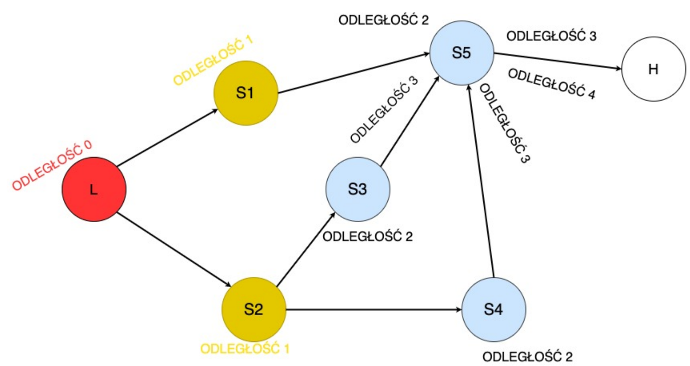
5.2.2 Najkrótsza droga

W poprzednim podrozdziale dowiedzieliśmy się, jak odwiedzić wszystkie osiągalne wierzchołki z wierzchołka **L**. Teraz przyjrzymy się odległości i scieżce od wierzchołka **L** do wierzchołka **L**. Wszystkie ścieżki wyprowadzone przez przeszukiwanie wszerz są najkrótszymi scieżkami od wierzchołka początkowego do końcowych wierzchołków.

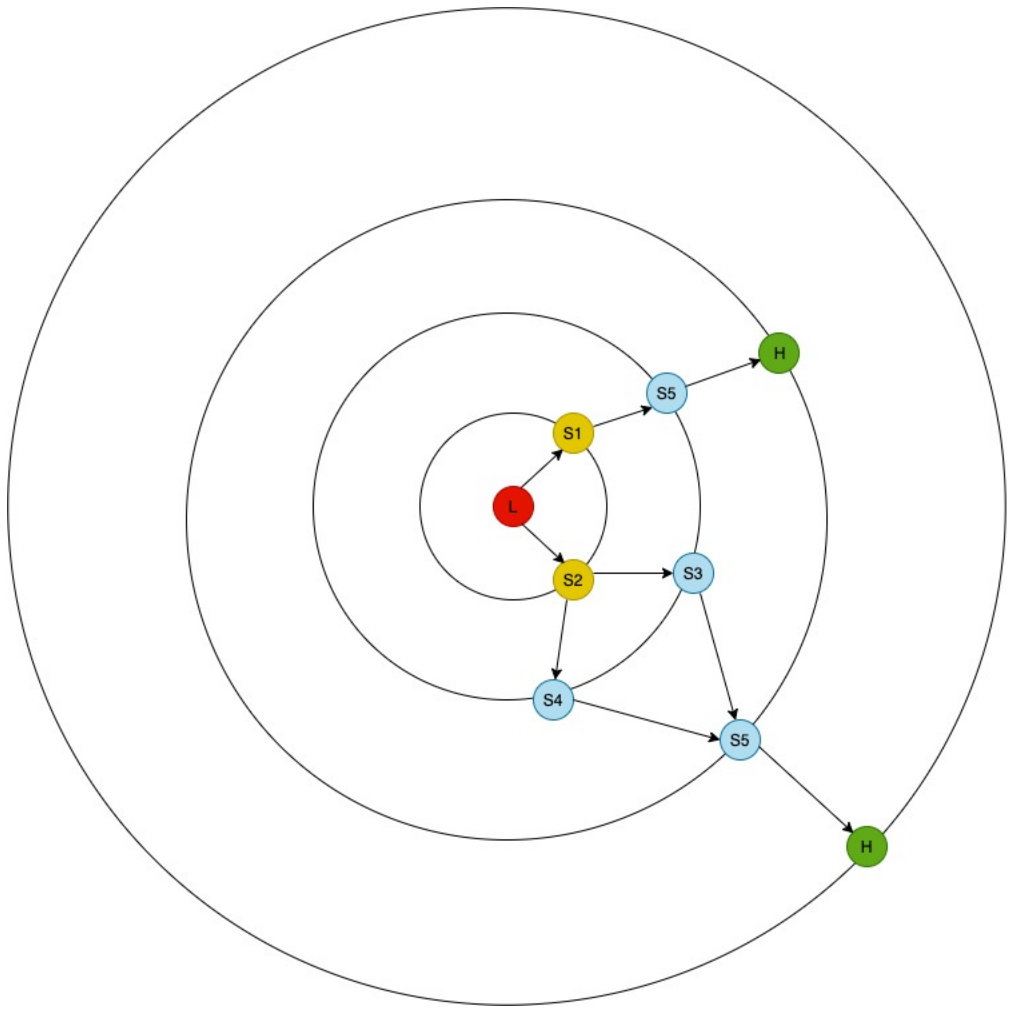
W przeszukiwaniu wszerz odwiedziliśmy najpierw wierzchołek **S1** oraz **S2**. Co oznacza, że najpierw dodajemy **S1** i **S2** do kolejki, a czas dodawania w **kroku 2** algorytmu jest taki sam. To mówi nam, że możemy odwiedzić wierzchołki **S1** i **S2** w odległości 1. Należy pamiętać, że w tym przypadku 1 odległość to 1 krawędź. Ścieżki od wirzchołka **L** do wierzchołków **S1** i **S2** to odpowiednio **L** do **S1** i **L** do **S2**. Ilustruje to poniższy graf.



Następnie wyodrębniliśmy wierzchołek **S1** z kolejki i dodaliśmy jego sąsiadów do kolejki. Teraz możemy dotrzeć z wierzchołka **L** do wierzchołka **S5** w 2 odległości. Dzieje się tak, ponieważ do wierzchołka **S1** możemy dodtrzeć z wierzchołka **L** w 1 odległość i do wierzchołka **S5** z wierzchołka **S1** w 1 odległość. Podobnie sprawa wygląda w przypadku wierzchołków **S3** i **S4**, które z kolei są sąsiadami wierzchołka **S2**. Dotarcie do wierzchołka **S5** z wierzchołków **S3**, **S4**, **H** zajmuje nam 3 odległości startując od wierzchołka **L**. Jeżeli zdecydowalibyśmy się obrać drogę **L** $\rightarrow$ **S2** $\rightarrow$ **S3** $\rightarrow$ **S5** $\rightarrow$ **H** - droga zajmie nam 4 odległości.



Scieżki wyprowadzone z przeszukiwania wszerz są najkrótszą ścieżką od wierzchołka do niektórych wierzchołków. Algorytm najpierw odwiedza wszystkie wierzchołki sąsiadujące z aktualnym wierzchołkiem, zanim przejdzie do następnego wierzchołka. Jeżeli wszystkie sąsiednie wierzchołki zostały odwiedzone, procedura zostaje powtórzona dla sąsiadów itd.. Możemy powiedzieć, że odwiedzamy wierzchołki wewnątrz okręgu o promieniu 1, 2 i tak dalej, jak obrazuje to poniższy rysunek.



L
S1
S2
S5
S3
S4
S5
H
H

5.2.3 Implementacja

5.2.3.1 Python

```
from collections import deque

def bfs(graph, vertex):
    queue = deque([vertex])
    level = {vertex: 0}
    parent = {vertex: None}
    while queue:
        v = queue.popleft()
        for n in graph[v]:
            if n not in level:
                queue.append(n)
                level[n] = level[v] + 1
                parent[n] = v

    return level, parent
```

1. Z modułu collections importujemy kolejkę deque
2. tworzymy funkcje bfs, która przyjmuje dwa parametry:
   - **graph** - graf, który będzie przeszukiwany
   - **vertex** - wierzchołek, z którego rozpoczynamy
3. inicjalizujemy zmienne:
   - **queue** - kolejka, do której będziemy dodawać sąsiadów odwiedzonego wierzchołka, na początku dodajemy do niej wierzchołek startowy
   - **level** - słownik, do którego jako klucze będziemy dodawać odwiedzone wierzchołki, a wartościami będzie liczba krawędzi, którą należy pokonać od wierzchołka startowego do wierzochołka, w którym obecnie się znajdujemy, początkowa wartość to wierzchołek startowy i 0
   - **parent** - słownik, którego kluczem będzie wierzchołek, w którym się obecnie znajdujemy, a wartością wierzchołek, z którego przybyliśmy, początkowa wartość to wierzchołek startowy i None
4. Pętla **while**, która działa dopóki kolejka (**queue**) nie jest pusta, w środku pętli wykonywane są następujące operacje
   - inicjalizujemy zmienna **v**, jako pierwszy element, który znajduje się w kolejce (kolejka jest **FIFO**, dlatego do wyodrębnienia elementu używana jest metoda **popleft**)
   - tworzymy pętle **for**, która iteruje po liście sąsiadów aktualnie badanego wierzchołka, zmienna **n** oznacza sąsiada, czyli elementem z listy sąsiedztwa przypisanej do wierzchołka **v**. W środku pętli wykonywane są czynności:

- Insturkcja warunkowa **if** sprawdza czy **n** (sąsiad) (nie)jest jednym z kluczy w słowniku level, czyli sprawdzamy czy sąsiad aktualnie badanego wierzchołka był odwiedzony.
- jeżeli nie był odwiedzony wykonujemy następujące czynności:
  * Na koniec kolejki dodajemy **n**, czyli aktualnie sprawdzany wierzchołek sąsiadujący z wierzchołkiem **v**
  * następnie w słowniki **level** tworzymy pare klucz:wartość, gdzie kluczem jest **n** - czyli badany sąsiad wierzchołka **v**, a wartością jest obiekt znajdujący się pod kluczem **v** w słowniku level, inkrementoway o 1 - czyli odległość (liczba krawędzi), jaką musimy pokonać, aby dodtrzec od wierzchołka startowego do **n**
  * ostatnią operacją jest stworzenie pary klucz:wartość w słowniku **parent**, który jako klucz przyjmuje aktualnie badanego sąsiada - **n**, a wartość wierzchołek, dla którego **n** jest sąsiadem, czyli **v**
- jeżeli aktualnie badany sąsiad jest już kluczem w tablicy **level**, nie podejmujemy żadnych czynności, tylko przechodzimy do kolejnej iteracji w pętli **for**

5. gdy kolejka jest pusta, funkcja zwraca zbiór, który zawiera w sobie słownik **level** oraz **parent**, po czym program kończy działanie.

5.2.3.2 Haskell

```
data Vertex = Vertex {
                        vertexLabel :: [Char]
                      , vertexNeighbors :: [[Char]]
                      , vertexDistance :: Int
                      , vertexPredecessor :: [Char]
                    } deriving Show

data Graph = Graph [Vertex] deriving Show

vertexInVertexes :: Vertex -> [Vertex] -> Bool
vertexInVertexes _ [] = False
vertexInVertexes Vertex {vertexLabel = label} (x:y) = foldl (\
↪   acc x -> vertexLabel x == label || acc ) False (x:y)

graphVertex :: Graph -> [[Char]] -> [Vertex]

graphVertex (Graph []) _ = []
graphVertex (Graph (x:y)) [] = x : y
graphVertexes (Graph (x:y)) keys = filter (\ z -> vertexLabel z
↪   `elem` keys) (x:y)



bfs :: Graph -> Graph -> [Vertex] -> [Vertex] -> Graph
bfs (Graph []) _ _ _ = Graph []
bfs _ outGraph [] _ = outGraph
bfs (Graph (a:b)) (Graph (c:d)) (e:f) (g:h) = bfs inGraph
↪   outGraph queue seen'
    where inGraph = Graph (a:b)
          eLabel = vertexLabel e
          eNeighbors = vertexNeighbors e
          eVertexNeighbors = graphVertexes inGraph eNeighbors
          dist = vertexDistance e + 1
          seen = g : h
          filteredNeighbors = filterVertexNeighbors seen
          ↪   eVertexNeighbors
          enqueue = updateDistPred filteredNeighbors dist eLabel
          outGraph = Graph $ (c:d) ++ enqueue
          queue = f ++ enqueue
          seen' = seen ++ enqueue


filterVertexNeighbors :: [Vertex] -> [Vertex] -> [Vertex]
```

```haskell
filterVertexNeighbors _ [] = []
filterVertexNeighbors [] _ = []

filterVertexNeighbors s vn = filter (\ x -> not $
↪  vertexInVertexes x s) vn
updateDistPred :: [Vertex] -> Int -> [Char] -> [Vertex]
updateDistPred [] _ _ = []
updateDistPred (x:y) dist perdLabel = map (\ (Vertex label n _ _
↪  ) -> Vertex label n dist perdLabel) (x:y)


printGraph :: Graph -> IO ()
printGraph (Graph []) = putStrLn ""
printGraph (Graph (x:y)) = do
    print x
    printGraph (Graph y)
    return ()

main :: IO ()
main = do
  let myGraph = Graph [
                  Vertex "L" ["S1", "S2"          ] 0 ""
                , Vertex "S1" ["S5"     ] 0 ""
                , Vertex "S2" ["S3", "S4"          ] 0 ""
                , Vertex "S3" ["S5"     ] 0 ""
                , Vertex "S4" ["S5"] 0 ""
                , Vertex "S5" ["H"     ] 0 ""
                , Vertex "H" [] 0 ""
                ]
  let queue = graphVertexes myGraph ["L"]
  let outGraph = Graph queue
  let seen = queue
  printGraph $ bfs myGraph outGraph queue seen
  return ()
```

1. Nasz typ Vertex uzupełniamy o dwa dodatkowe pola: **vertexDistance -** odległośc, jaka dzieli nas od wierzchołka startowego oraz **vertexPredecessor -** czyli rodzic wierzchołka aktualnie sprawdzanego

2. następnie tworzymy funkcje, tzw. "gettery"

   - funkcja **vertexInVertexes** na wejściu przyjmuje węzeł, listę węzłów oraz zwraca prawdę lub fałsz:
     - jeżeli podana lista węzłów jest pusta, zwraca fałsz
     - Następnie za pomocą funkcji foldl redukujemy liste wierzchołków do wartości logicznej
       - funkcje z rodziny fold przetwarzają uporządkowane kolekcje danych w celu budowania końcowego wyniku przy pomocy jakieś funkcji łączącej elementy
       - jako parametry wywołania funkcji foldl przekazujemy anonimową funkcje, którą tworzymy za pomocą \, która sprawdza w warunku logicznym OR czy x jest równy etykiecie wierzchołka, do funkcji przekazujemy zaprzeczenie listy, która posiada dwa elementy
     - jeżeli conajmniej jeden identyfikator wierzchołka na liście jest zgodny z identyfikatorem wierzchołka wejściowego, wynik będzie prawdą
     - funkcja **graphVertexes** przyjmuje graf, listę łańchchów znaków i zwraca listę wierzchołków
       - jeżeli graf jest pusty, zwracamy pustą listę
       - jeżeli lista wierzchołków jest pusta, zwracamy listę znajdującą sięw grafie
       - używamy funkcji filter, która znowu za pomocą funkcji anonimowej i funkcji elem sprawdza, czy etykieta wierzchołka znajduje się w liście etykiet (**keys**)
   - tworzymy funkcje bfs, która jest implementacją naszego algorytmu
     - funkcja przyjmuje następujące parametry:
       - graf, w którym bedziemy przeprowadzac bfs
       - graf, który będzie grafem wyjściowym
       - kolejke
       - liste odwiedzonych wierzchołków
     - **eLabel** pobiera identyfikator obecnie badanego wierzchołka
     - **eNeighbors** pobiera liste identyfikatorów wierzchołków sąsiadujących
     - **eVertexNeighbors** pobiera wierzchołki na podstawie etykiet sąsiadów

  - **dist** ustawia odległość sąsiadujących wierzchołków na o jeden większą od wierzchołka aktualnie analizowanego **seen** - wierzchołki, które były wczesniej w kolejece sa oznaczone jako odwiedzone
  - **filteredNeighbors** usuwa wszystkie sąsiadujące wierzchołki, które były już odwiedzone
  - **enqueue** aktualizuje etykietę rodzica i odległość dla każdego sąsiada wierzchołka
  - **outGraph** aktualizuje nasz graf
  - **queue** dodaje sąsiadów do kolejki
  - **seen'** aktualizacja kolejki odwiedzonych wierzchołków
- funkcje pomocnicze:
  - **filterVertexNeighbors** funkcja wyrzuca z listy sąsiadów wszystkie wierzchołki, które zostały odwiedzone
  - **updateDistPred** przechodzi graf i aktualizuje wartości rodzica oraz dystans

Program zwraca w terminalu wynik działania algorytmu bfs:

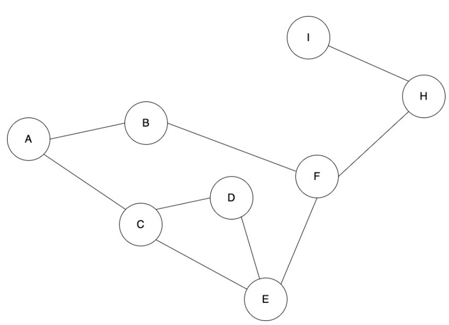
```
Vertex {vertexLabel = "S1", vertexNeighbors = ["S5"], vertexDistance = 1, vertexPredecessor = "L"}
Vertex {vertexLabel = "S2", vertexNeighbors = ["S3","S4"], vertexDistance = 1, vertexPredecessor = "L"}
Vertex {vertexLabel = "S5", vertexNeighbors = ["H"], vertexDistance = 2, vertexPredecessor = "S1"}
Vertex {vertexLabel = "S3", vertexNeighbors = ["S5"], vertexDistance = 2, vertexPredecessor = "S2"}
Vertex {vertexLabel = "S4", vertexNeighbors = ["S5"], vertexDistance = 2, vertexPredecessor = "S2"}
Vertex {vertexLabel = "H", vertexNeighbors = [], vertexDistance = 3, vertexPredecessor = "S5"}
```

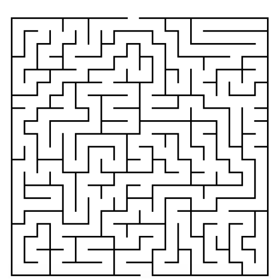
5.3 Przeszukiwanie w głąb

Przeszukiwanie w głąb jest procedura, która ujawnia wiele cennych informacji o grafie. Najprostszym pytaniem, do którego się odnosi jest:

Do jakich części grafu można dotrzeć z danego wierzchołka?

5.3.1 Zasada działania algorytmu





Aby zrozumieć ten problem, postawmy się na miejscu komputera, który dostaje do przeanalizowania graf zaimplementowany w formie listy sąsiedztwa. Reprezentacja ta pozwala nam na znalezienie sąsiadów wierzchołka. Posiadając tylko tę operację eksploratacja grafu przypomina przechodzenie labiryntu. Startujemy z ustalonego miejsca i za każdym razem, gdy dojdziemy do skrzyżowania (wierzchołka), istnieje wiele dróg, którymi możemy podążać. Zły wybór może nas poprowadzić dookoła lub spowodować przeoczenie dostępnej części labiryntu. Oczywiście podczas eksploratacji musimy zapamiętać pewne informacje.

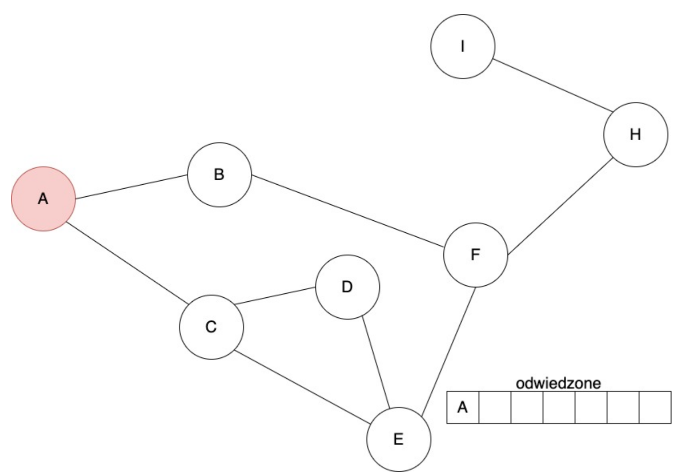
Najprostszym sposobem, aby przejść labirynt jest użycie kredy i sznurka. Zaznaczając skrzyżowania, które już odwiedziliśmy kredą - zapobiegamy zapętlaniu się. Sznurek pomoże nam wrócić do miejsca, z którego przyszliśmy, co umożliwia przejście do niezbadanych fragmentów.

Jak zasymulować kredę i sznurek na komputerze? Oznaczenia kredą będą reprezentowane za pomocą przypisania każdemu wierzchołkowi wartości logcznej (**True**), jeżeli został on już odwiedzony. Cyberanalogią sznurka będzie **stos**, czyli **kolejka LIFO** (*ang.* ***L**ast **I**n **F**irst **O**ut*). Rolą sznurka jest oferowanie dwóch operacji - rozwijanie, aby dostać się do kolejnego wierzchołka (czyli dodanie wierzchołka na stos) oraz zwijanie, aby powrócić do poprzedniego (czyli zdejmowanie wierzchołka ze stosu).

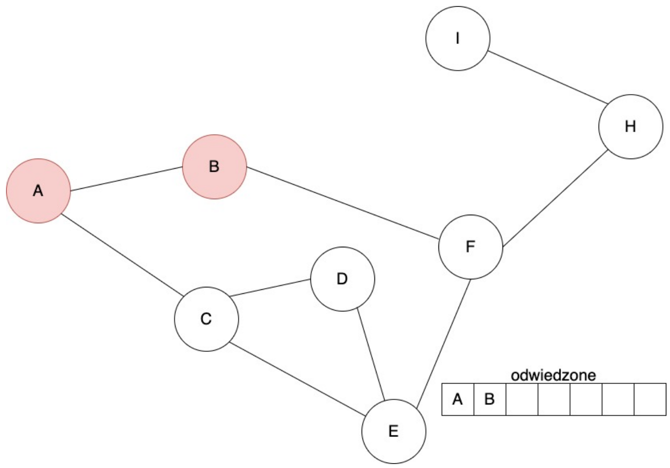
W naszej implementacji zamiast jawnie tworzyć i utrzymywać stos, wykorzystamy rekurencje, która jest implementowana przy użyciu rekordów stosu aktywacji.

5.4 Algorym przeszukiwania w głąb

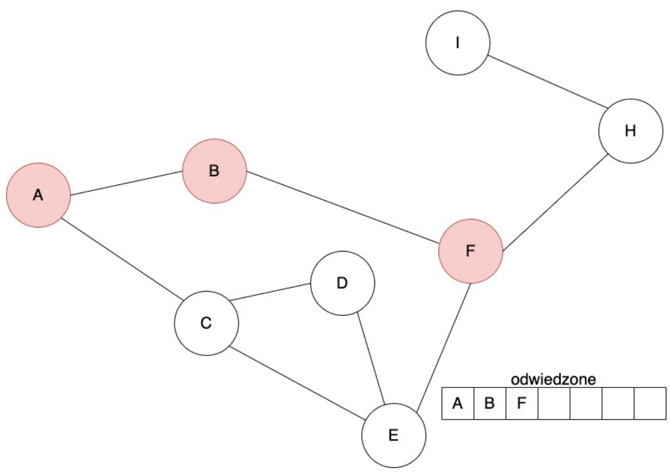
Sprawdźmy, jak algorytm działa. Na poniższym rysunku szukanie zaczynamy od wierzchołka **A**. Stan początkowy będzie następujący



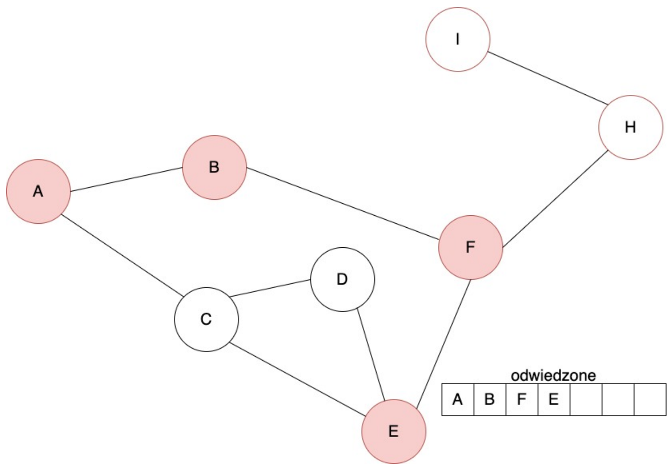
Następnie idziemy do wierzchołka **B** i zaznaczamy go jako *odwiedzony*. Warto zauważyć,że odwiedzamy wierzchołki w kolejności alfabetycznej. Dlatego pomijamy wierzchołek **C**.



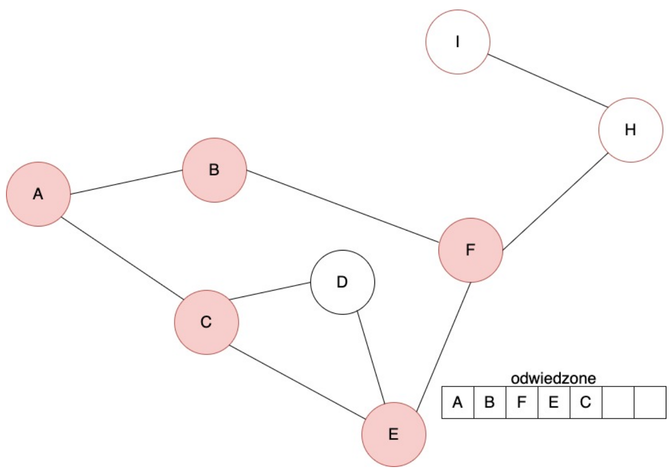
Następnie mamy do wyboru dwie drogi - wierzchołek **A** i wierzchołek **F**. Ponieważ **A** był odwiedzony idziemy do wierzhcołka **F**. Oznaczmy wierzcohłek **F** jako odwiedzony



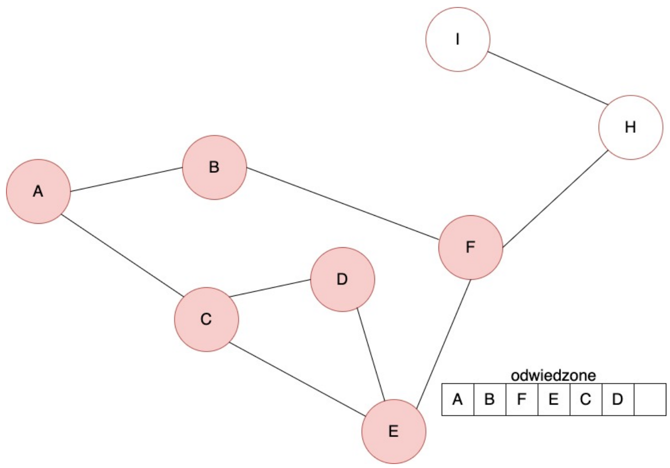
Z wierzchołka **F** podążamy według kolejności alfabetycznej, a zatem idziemy na wierzchołek **E** i oznaczamy go jako odwiedzony.



Sprawa wygląda podobnie w wierzchołku **E** - patrzymy na sąsiadów i wybieramy w kolejności alfabetycznej. Przechodzimy zatem do wierzchołka **C** i oznaczamy go jako odwiedzony.



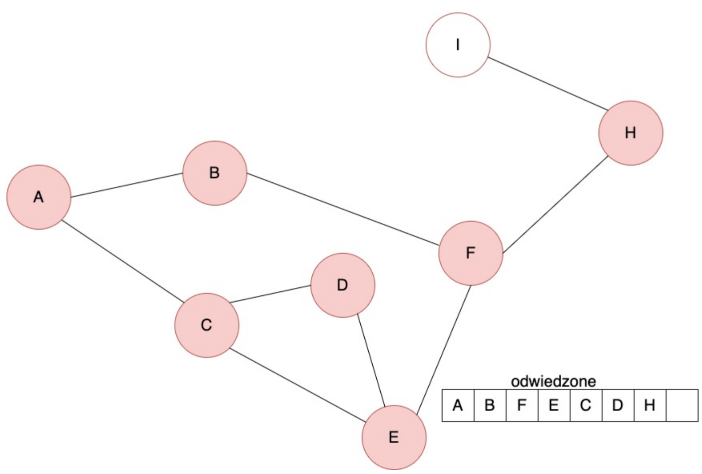
Z wierzchołka **C** przechodzimy na wierzchołek **D** i zaznaczamy go jako odwiedzony.



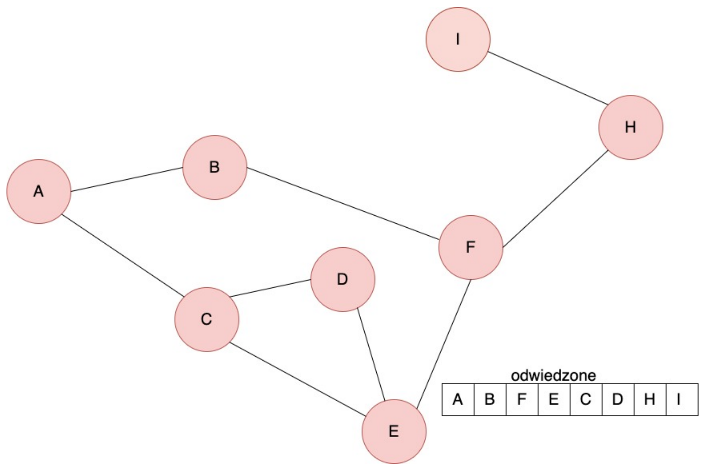
Teraz dotaliśmy do ślepego zaułka. Ponieaż wszystkie sąsiednie wierzchołki **D** zostały odwiedzone, musimy wracać jak po sznurku do wierzchołków, z których przybyliśmy i patrzeć na wierzchołki, których nie eksplorowaliśmy do tej pory. Zatem idziemy następującą drogą

$$\mathbf{D} \rightarrow \mathbf{C} \rightarrow \mathbf{E} \rightarrow \mathbf{F}$$

Jesteśmy na wierzchołku **F** - przechodzimy na wierzchołek **H** i oznaczamy jako odwiedzony.



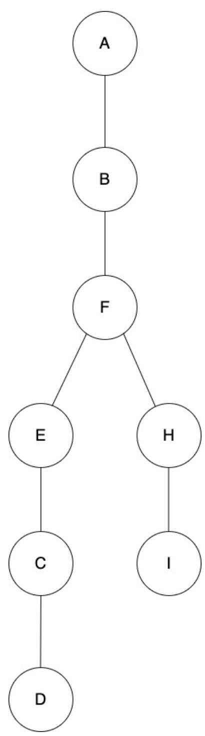
Został nam ostatni nieodwiedzony wierzcohłek **I**. Przechodzimy do niego i zaznaczamy, jako odwiedzony.



Tym oto sposobem odwiedziliśmy wszystkie wierzchołki.

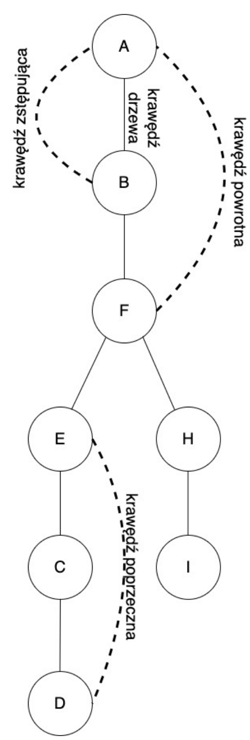
5.5 Klasyfikacja krawędzi w przeszukiwaniu w głąb

Scieżka, którą otrzymaliśmy za pomocą algorytmu przeszukiwania w głąb układa się w strukture drzewiastą. Gdy zastosujemy algorytm przeszukiwania w głąb dla grafu z powyższego podroździału, otrzymamy następujące drzewo.



Klasyfikujemy krawędzie na cztery rodzaje

- **krawędzie drzewowe** - krawędzie reprezentujące wywołania rekurencyjne
- **krawędzie powrotne** - krawędzie prowadzące do przodka danego wierzchołka w DFS
- **krawędzie zstępujące** - krawędzie do potomka danego wierzchołka w DFS
- **krawędzie poprzeczne** - pozostałe krawędzie (nie prowadzą ani do potomka, ani do przodka w drzewie DFS)



5.6 Implementacja

5.6.1 Python

```
def dfs(graph, vertex):
    parents = {vertex: None}
    dfs_visit(graph, vertex, parents)


def dfs_visit(graph, vertex, parents):
    for n in graph[vertex]:
        if n not in parents:
            parents[n] = vertex
            dfs_visit(graph, n, parents)
```

1. Rozpoczynamy przeszukiwanie w głąb wraz z inicjalizacją słownika **parents**, do którego jako klucz zapisujemy wierzchołek startowy, a jako wartość **None**

   - Zarządzamy odwiedzonymi wierzchołkami za pomocą kluczy i wartości w strukturze **parents**. Nazywamy ten słownik **parents**, ponieważ zarządza on węzłami nadrzędnymi w strukturze drzewa **DFS**. To znaczy **parents[klucz]** zwraca nadrzędny węzeł (jakiś wierzchołek grafu) węzła, który jest kluczem (również jakimś wierzchołkiem).

2. **dfs_visit** jest głównym procesem przeszukiwania w głąb

   - W tej operacji wyciągamy sąsiadów danego wierzchołka, sprawdzamy czy wyądrębniony sąsiad został odwiedzony. Jeżeli nie, wywołujemy na nim funckje **dfs_visit**
   - Zauważmy, że **graph[vertex]** zwraca sąsiadów danego wierzchołka. Kiedy rekurencyjne wywołanie **dfs_visit** jest skończone, powracamy do pętli **for** i powtarzamy te operację dla następnych sąsiadów. Po zakończeniu pierwszego wywołania pętli, możemy odtworzyć scieżkę **DFS** od rodziców.

5.6.2 Haskell

```
depthFirst :: Eq a => a -> Graph a -> [a]
depthFirst root (G _nodes edges)
  = reverse $ go [] root
  where
    go seen x
      | x `elem` seen = seen
      | otherwise = foldl' go (x:seen) (edges x)
```

Jak działa kod?

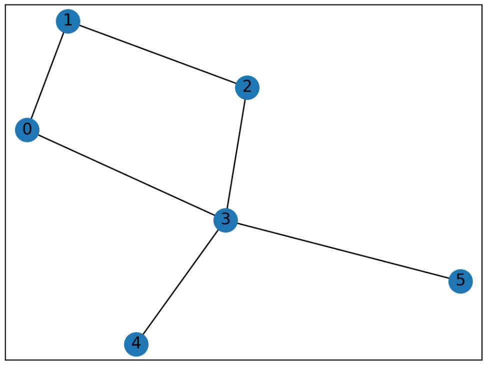
1. Używamy funkcji **foldl'**, ponieważ przechodzimy wierzchołki od lewej do prawej
2. **foldl'** rozpoczyna się od bieżącego wierzchołka poprzedzonego listą odwiedzonych wierzchołków (na początku pusta lista)
3. Następnie od lewej do prawej, odwiedza każdy wierzochłek dostępny bezpośrednio z bieżącego wierzchołka
4. W każdym ńastępnym"wierzchołku buduje odwrotną kolejność w głąb poddrzewa plus widziane już węzły.
5. Widoczne już wierzchołki są przenoszone do każdego ńastępnego"wierzchołka (w kolejności od lewej do prawej).
6. Jesli nie ma dostępnych wierzchołków z bieżącego wierzchołka, zwraca tylko bierzący wierzchołek dołączony do listy wszystkich widzianych wierzchołków.

6 Wizualizacja grafów

6.1 Python

```
import matplotlib.pyplot as plt
import networkx as nx


class GraphVisualization:
    def __init__(self):
        self.visual = []

    def add_edge(self, a, b):
        self.visual.append([a, b])

    def visualize(self):
        G = nx.Graph()
        G.add_edges_from(self.visual)
        nx.draw_networkx(G)
        plt.show()


if __name__ == "__main__":
    G = GraphVisualization()
    G.add_edge(0, 3)
    G.add_edge(1, 0)
    G.add_edge(1, 2)
    G.add_edge(2, 1)
    G.add_edge(3, 2)
    G.add_edge(3, 4)
    G.add_edge(3, 5)
    G.visualize()
```

1. importujemy bibliotekę matplotlib jako plt
2. importujemy biblioteke networkx jako nx
3. Tworzymy klase **GraphVisualization** - w jej konstruktorze tworzymy listę **visual**, która przechowuje krawędzie tworzące graf
4. metoda **addEdge** wprowadza wierzchołki krawędzi i dołącza je do listy **visual**
5. w metodzie **visualize** G jest nowa instancją klasy nx.graph, która dostarcza nam narzędzi aby utworzyć graf i go narysować.



Rysunek 8: Wizualizacja grafu Python

6.2 Haskell

```haskell
import Data.GraphViz

graph :: DotGraph Int

graph = graphElemsToDot graphParams nodes edges

graphParams :: GraphvizParams Int String Bool () String

graphParams = defaultParams

nodes :: [(Int, String)]

nodes = map (\x -> (x, "")) [1..4]

edges:: [(Int, Int, Bool)]

edges = [ (0,3, True)
        , (1, 0, True)
        , (1, 2, True)
        , (2,1, True)
        , (3, 2, True)]

main = addExtension (runGraphviz graph) Png "graph"
```

1.

Literatura

[1] Lech Banachowski, Krzyszyof Dikis, Wojciech Rytter *Algorytmy i struktury danych*. Wydawnictwo Naukowe PWN SA, Warszawa, 2018

[2] Alejandro Serrano Mena *Practical Haskell A Real World Guide to Programming*. Apress Media, Califronia, 2019

[3] Aditya Y. Bhargava *Grokking Algorithms: An Illustrated Guide for Programmers and Other Curious People* . Manning Publications, Nowy York, 2017

[4] MIT OpenCourseWare *https://www.youtube.com/watch?v=s-CYnVz-uh4*. 2013

[5] *https://lettier.github.io/posts/2016-04-29-breadth-first-search-in-haskell.html*

[6] *https://en.wikipedia.org/wiki/Fold_(higher-order_function)*

[7] S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani *Algorithms* UC Berkeley, 2006