Grafy - *Haskell*

Programowanie funkcyjne

Adrianna Saribekyan, Ewelina Kowal, Rafał Bocul

29 stycznia 2022

Spis treści

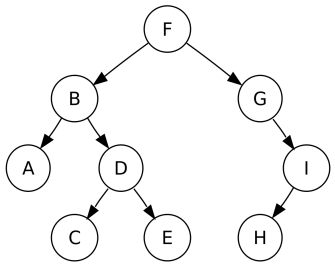
1 Wiadomości wstępne

1.1 Grafy

Grafy są podstawową strukturą danych do reprezentowania sieci. Pozwalają na większą dowolność w przedstawianiu struktur niż drzewa.

1.2 Cel

Przedstawimy przykładowe implementacje kodów angażujących grafy w języku programowania *Haskell*, który posiada wiele funkcji umożliwiających pracę z nimi.

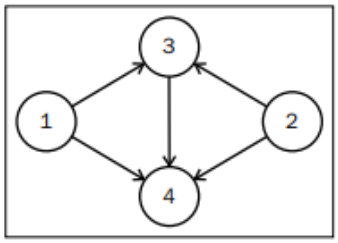


Rysunek 1: Przykładowy graf

2 Przedstawienie grafu za pomocą list krawędzi

Graf może być zdefiniowany za pomocą **list krawędzi**. Jest to struktura danych używana do reprezentowania grafu jako list jego krawędzi, gdzie krawędzie są skończonymi listami składającymi się z wierzchołka początkowego oraz końcowego.

Do stworzenia kodu używamy biblioteki *Data.Graph*. Wierzchołek (węzeł) to *Inc.*, a *buildG*, to funkcja służąca do skonstruowania struktury danych grafu z listy krawędzi.



Rysunek 2: Graf wykorzystywany w programach list krawędzi, list sąsiedztwa, przeszukiwaniu w głąb.

```haskell
import Data.Graph ( buildG, edges, vertices, Graph )
import Data.Graph

myGraph :: Graph

myGraph = buildG bounds edges
    where bounds = (1,4)
          edges = [ (1,3), (1,4), (2,3), (2,4), (3,4) ]

main :: IO ()
main = do
    print $ "The edges are " ++ (show.edges) myGraph
    print $ "The vertices are " ++ (show.vertices) myGraph
```

Rysunek 3: Implementacja kodu przedstawiającego graf za pomocą listy krawędzi.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\ANI> stack runhaskell "c:\Users\ANI\fizyka_techniczna\Vsem\Programowanie\Ada-1_3_4-Working\1\listofedges.hs"
"The edges are [(1,4),(1,3),(2,4),(2,3),(3,4)]"
"The vertices are [1,2,3,4]"
PS C:\Users\ANI>
```

Rysunek 4: Wynik kodu przedstawiającego graf za pomocą listy krawędzi.

3 Przedstawienie grafu za pomocą list sąsiedztwa

Kolejnym sposobem jest użycie **list sąsiedztwa**. Jest to zbiór nieuporządkowanych list służących do reprezentowania skończonego grafu.

Tutaj również korzystamy z paczki *Data.Graph*. Posłuży nam do czytania mapowania wierzchołka do listy połączonych wierzchołków. Wykorzystamy również funkcję *graphFromEdges'*, która zwraca skończoną listę złożoną z elementów- struktury danych grafu i mapowania numeru wierzchołka do jego klucza.

```haskell
import Data.Graph ( edges, graphFromEdges', vertices, Graph )
import Data.Graph

myGraph::Graph

myGraph=fst $ graphFromEdges' [ ("Node 1", 1, [3, 4]), ("Node 2", 2, [3, 4]), ("Node 3", 3, [4]), ("Node 4", 4, []) ]

main :: IO ()
main = do
    putStrLn $ "the edges are "++ (show.edges) myGraph
    putStrLn $ "The vertices are "++ (show.vertices) myGraph
```

Rysunek 5: Implementacja kodu przedstawiającego graf za pomocą list sąsiedztwa.

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\ANI> stack runhaskell "c:\Users\ANI\fizyka_techniczna\Vsem\Programowanie\graphsHaskell\adjencylist.hs"
the edges are [(0,2),(0,3),(1,2),(1,3),(2,3)]
The vertices are [0,1,2,3]
PS C:\Users\ANI>
```

Rysunek 6: Wynik kodu przedstawiającego graf za pomocą list sąsiedztwa.

4 Sortowanie topologiczne przeprowadzone na grafie

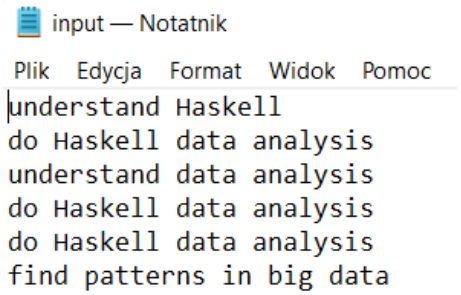
Jeśli graf jest skierowany (jego krawędzie mają kierunek), to jednym z naturalnych jego porządków jest **sortowanie topologiczne**, gdzie każdy wierzchołek poprzedza wszystkie te wierzchołki, do których prowadzą wychodzące od niego krawędzie.

W podejściu opartym na sieciach zależności, topologiczne sortowanie pokaże możliwe kolejności wyliczenia wszystkich wierzchołków, które spełniają takie zależności.

Tutaj użyjemy wbudowanej funkcji Haskella *topSort*, która pozwala na ten rodzaj sortowania elementów grafu. Stworzymy graf zależności i wybierzemy kolejność za pomocą sortowania topologicznego.

```
    import Data.Graph ( buildG, topSort, Graph )
 1  import Data.Graph
 2  import Data.Map (Map, (!), fromList)
 3  import Data.List (nub)
 4
    main :: IO ()
 5  main = do
 6  |  ls <- fmap lines getContents
 7  |  let g = graph ls
 8  |  putStrLn $ showTopSort ls g
 9
10  graph :: Ord k => [k] -> Graph
11
12  graph ls = buildG bounds edges
13  |  where bounds = (1, (length.nub) ls)
14  |  |  |  |  edges = tuples $ map (mappingStrToNum !) ls
15  |  |  |  |  mappingStrToNum = fromList $ zip (nub ls) [1..]
16  |  |  |  |  tuples (a:b:cs) = (a, b) : tuples cs
17  |  |  |  |  tuples _= []
18
19  showTopSort :: [String] -> Graph -> String
20
21  showTopSort ls g =
22  |  |  unlines $ map (mappingNumToStr !) (topSort g)
23  |  |  where mappingNumToStr = fromList $ zip [1..] (nub ls)
```

Rysunek 7: Implementacja kodu przedstawiającego sortowanie topologiczne przeprowadzone na grafie.



Rysunek 8: Plik tekstowy użyty do kodu przedstawiającego sortowanie topologiczne przeprowadzone na grafie.

Powyższy tekst w pliku na rysunku 8 powinniśmy rozumieć jako:

- należy zrozumieć *Haskella*, aby przeprowadzić analizę danych w *Haskellu*,
- należy zrozumieć analizę danych, aby przeprowadzić analizę danych w *Haskellu*,
- należy rzeprowadzić analizę danych w *Haskellu*, aby znaleźć wzór w *big data* (duże zmienne i różnorodne zbiory danych, których przetwarzanie i analiza jest trudna).

```
rafcio@rafcio-M-Linux:~/Pulpit/Haskell_projects/Programs/3$ runhaskell topolog
alsort.hs < input.txt
understand data analysis
understand Haskell
do Haskell data analysis
find patterns in big data
```

Rysunek 9: Wynik kodu przedstawiającego sortowanie topologiczne przeprowadzone na grafie.

5 Badanie grafu metodą przeszukiwania w głąb (DFS)

Kolejnym rozwiązaniem jest przeszukiwanie grafu za pomocą metody **DFS**, czyli **przeszukiwania w głąb**. Polega ona na przebadaniu każdej krawędzi, która wychodzi z danego wierzchołka, po czym wraca do wierzchołka "początkowego", czyli takiego, z którego przeszliśmy do wspomnianego wyżej, danego wierzchołka.

Implementacja sortowania topologicznego, algorytmu rozwiązywania labiryntu, czy znajdowania połączonych komponentów są przykładami algorytmów wykorzystujących przeszukiwanie w głąb.

```haskell
import Data.Graph
import Data.Array ((!))

graph :: (Graph, Vertex -> (Int, Int, [Int]))
graph = graphFromEdges' [ (1, 1, [3, 4] )
                        , (2, 2, [3, 4])
                        , (3, 3, [4])
                        , (4, 4, []) ]

main = do
  print $ depth graph 0

depth :: GHC.Ix.Ix t => (GHC.Arr.Array t [t], t -> (a1, a2, c)) -> t -> [a2]
depth g i = depth' g [] i
depth' g@(gShape, gMapping) seen i =
  key : concat (map goDeeper adjacent)
  where goDeeper v = if v `elem` seen then [] else depth' g (i:seen) v
        adjacent = gShape ! i
        (_, key, _) = gMapping i
```

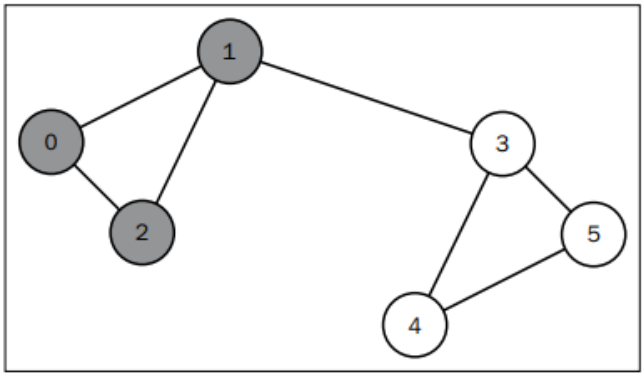
Rysunek 10: Implementacja kodu przedstawiającego badanie grafu metodą przeszukiwania w głąb (DFS).

```
rafcio@rafcio-M-Linux:~/Pulpit/Haskell_projects/Programs/4$ ghc -o Depth-First.x
 depth-first.hs
[1 of 1] Compiling Main             ( depth-first.hs, depth-first.o )
Linking Depth-First.x ...
rafcio@rafcio-M-Linux:~/Pulpit/Haskell_projects/Programs/4$ ./Depth-First.x
[1,3,4,4]
```

Rysunek 11: Wynik kodu przedstawiającego badanie grafu metodą przeszukiwania w głąb (DFS).

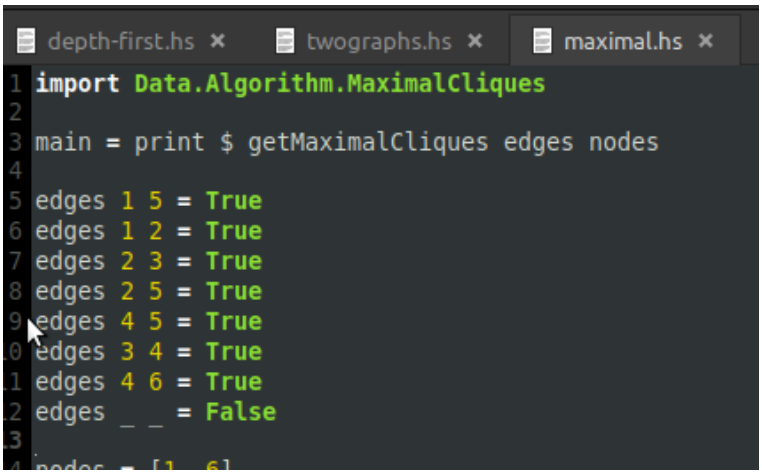
6 Znajdowanie maksymalnych klik na wykresie

Haskell oferuje takie udogodnienia jak biblioteki grafów witalnych, z których jedna z nich to biblioteka wykrywania z **Data.Algoritm.Maximual Cliques**. Klika na wykresie to podgraf, w którym wszystkie węzły mają połączenia między sobą i jest przedstawiony następująco:

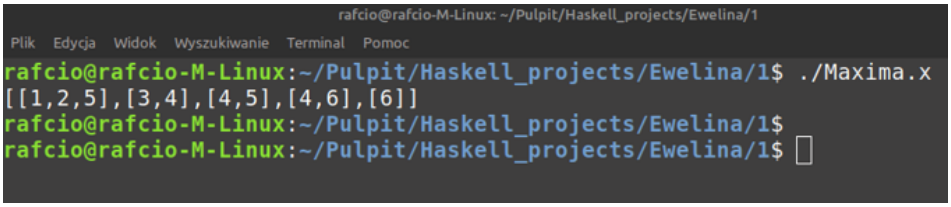


Rysunek 12: Graf przedstawiający dwie połączone ze sobą kliki.

Wykres ten zawiera dwie kliki. Może on przedstawiać połączone ze sobą strony internetowe, co możemy wywnioskować ze względu na strukturę wykresu. Wraz ze wzrastającą liczbą połączeń, trudniejsze staje się znalezienie największej kliki.



Rysunek 13: Kod zawierający bobliotekę Data.Algoritm.MaximualCliques.

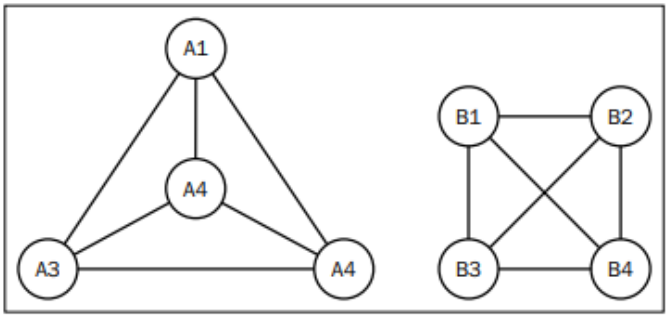


Rysunek 14: Wynik kodu zawierającego, który wyszukuje i sortuje największe kliki w grafie.

Jako informację zwrotną, otrzymujemy posegregowane poszczególne podgrupy połaczone ze sobą w naszym grafie. Algorytm sortuje kliki od największej do najmnniejszej i wypisuje je w programie.

7 Ustalanie izomorficzności dwóch wykresów.

Analizując różne sieci graficzne jesteśmy w stanie stwierdzić czy są one izomorficzne, czyli czy ich połączenia mają identyczny wzór. To pomaga nam odkryć, kiedy dwa pozornie różne grafiki sieci kończą tym samym mapowaniem sieci.



Rysunek 15: Przykładowe grafy, którymi będziemy się zajmować.

Do wykrycia izomorficzności grafów użyjemy funkcji **isIsomorphic** z **Data.Graph.Automorphism**. Biblioteka ta pomoże nam zbadać ewentualną identyczność połączeń powyższych grafów.

depth-first.hs × twographs.hs ×

```
1  import Data.Graph
2  import Data.Graph.Automorphism
3
4  graph = buildG (0,4) [ (1, 3), (1, 4)
5                       , (1, 2), (2, 3)
6                       , (2, 4), (3, 4) ]
7
8  graph' = buildG (0,4) [ (3, 1), (3, 2)
9                        , (3, 4), (4, 1)
10                       , (4, 2), (1, 2) ]
11
12 main = print $ isIsomorphic graph graph'
```

Rysunek 16: Wynik kodu zawierający funkcję Data.Graph.Automorfizm.

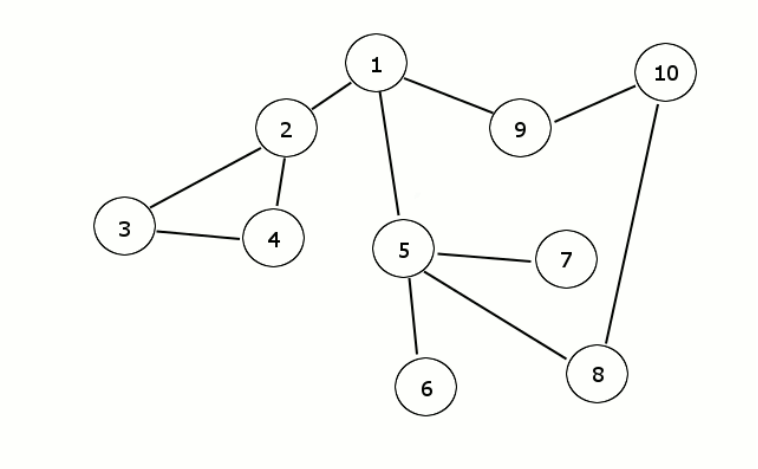
```
rafcio@rafcio-M-Linux: ~/Pulpit/Haskell_projects/Ewelina/2
Plik  Edycja  Widok  Wyszukiwanie  Terminal  Pomoc
rafcio@rafcio-M-Linux:~/Pulpit/Haskell_projects/Ewelina/2$ ./TwoGraphs.x
True
rafcio@rafcio-M-Linux:~/Pulpit/Haskell_projects/Ewelina/2$
```

Rysunek 17: Wynik kodu, który bada izomorficzność grafów.

Efektem działania kodu jest przebadanie grafów pod względem podobieństwa. Jako informację zwrotną otrzymujemy "True" lub "False", w zależności od tego, czy grafy są identyczne. Jak widać grafy, które badaliśmy są izomorficzne, stąd wynik "True".

8 Badanie grafu metodą przeszukiwania w głąb (BFS)

Metoda przeszukiwania grafu w szerz jest jednym z najprostszych algorytmów przeszukiwania grafu. Przeszukiwanie grafu rozpoczyna się od zadanego przez Nas wierzchołka i polega na odwiedzeniu wszystkich osiągalnych z niego wierzchołków. Za pomocą przeszukiwania grafu w szerz można wyznaczyć najkrótsze pod względem liczby krawędzi (ale nie wag) ścieżki między wierzchołkami źródłowymi, a pozostałymi wierzchołkami.



Rysunek 18: Metoda BFS

mainBreadth.hs

```haskell
import Data.Graph
import Data.Array ((!))

graph :: Graph
graph = buildG bounds edges
    where bounds = (1,7)
          edges = [ (1,2), (1,5)
                  , (2,3), (2,4)
                  , (5,6), (5,7)
                  , (3,1) ]

breadth g i = bf [] [i]
    where bf :: [Int] -> [Int] -> [Int]
          bf seen forest | null forest = []
                         | otherwise   = forest ++
                                         bf (forest ++ seen)
                                            (concat (map goDeeper forest))
            where goDeeper v = if elem v seen
                               then [] else (g ! v)
main = do
    print $ breadth graph 1
```

Rysunek 19: Kod BFS.

```
Plik  Edycja  Widok  Wyszukiwanie  Terminal  Pomoc
rafcio@rafcio-M-Linux:~/Pulpit/Haskell_projects/1.breadth-first$ ./Breadth.x
[1,5,2,7,6,4,3,1]
rafcio@rafcio-M-Linux:~/Pulpit/Haskell_projects/1.breadth-first$
```

Rysunek 20: Rezultat BFS.

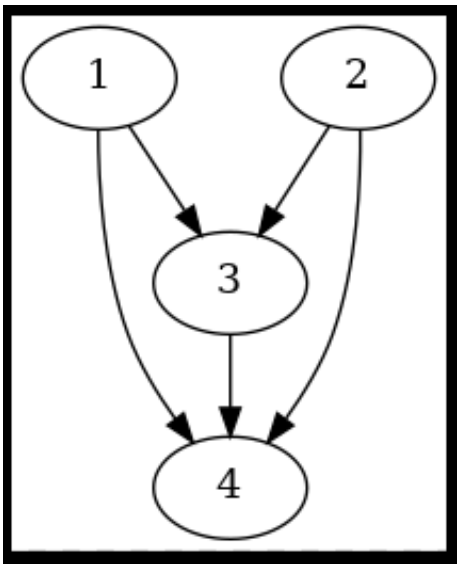
9 Przedstawienie grafu przy pomocy Graphviz

Przy pomocy biblioteki **graphvic** możemy łatwo narysować interesujące nas grafy. W świecie analizy danych wizualna interpretacja może ujawnić szczegóły, których ludzkie oko nie byłoby w stanie dostrzec.

graphviz.hs

```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 = [ (1, 3, True)
        ,(1, 4, True)
      , (2, 3, True)
      , (2, 4, True)
      , (3, 4, True)]

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

Rysunek 21: Kod GraphViz.



Rysunek 22: Rezultat GraphViz.

10 Stosowanie acyklicznych skierowanych grafów słów (DAWG)

Skierowany acykliczny wykres słów (DAWG) to struktura danych reprezentująca zestaw ciągów, często używana jako kompaktowy sposób przechowywania słownika. Każde słowo zaczyna się od węzła źródłowego i kończy w węźle ujścia. Pomiędzy źródłem, a ujściem znajduje się sieć węzłów, połączonych skierowanymi krawędziami oznaczonymi literą lub jako EOW (koniec słowa) podczas łączenia się z ujściem. Każdy węzeł ma co najwyżej jedną krawędź wychodzącą dla każdej litery lub EOW. Podążanie każdą możliwą ścieżką od węzła źródłowego do węzła ujścia daje listę słów w słowniku.

dawg.hs

```haskell
import qualified Data.DAWG.Static as D

import Data.Char (toLower, isAlphaNum, isSpace)
import Data.Maybe (isJust)

main = do
    let url = "http://norvig.com/big.txt"
    body <- simpleHTTP (getRequest url) >>= getResponseBody

    let corp = corpus body
    print $ isJust $ D.lookup "hello" corp
    print $ isJust $ D.lookup "goodbye" corp

getWords :: String -> [String]

getWords str = words $ map toLowe wordlike
    where wordlike =
            filter (\x -> isAlphaNum x || isSpace x) str

corpus :: String -> D.DAWG Char () ()
corups str = D.fromLang $ getWords str

```

Rysunek 23: Kod DAWG.

Niestety, ale ze względu na problemy z użyciem biblioteki HTTP nie udało nam się poprawnie zaimplementować kodu, a co za tym idzie nie mogliśmy ujrzeć rezultatu jego działania.

11 Praca z sieciami sześciokątnymi i kwadratowymi

Czasami wykres, z którym mamy do czynienia ma konkretną strukturę taką jak siatka kwadratowa lub heksagonalna. Wiele gier wideo używa heksagonalnego układu siatki, aby ułatwić ruch po przekątnej, ponieważ poruszanie się po przekątnej w kwadratowej siatce bardziej komplikuje wartości przebytego dystansu. Z drugiej strony kwadratowe struktury siatki są często używane w algorytmach manipulacji obrazem np. wypełnianie powodziowe.

hexa.hs

```haskell
import Math.Geometry.Grid (indices, neighbours)
import Math.Geometry.Grid.Hexagonal (hexHexGrid)
import Math.Geometry.Grid.Square (rectSquareGrid)
import Math.Geometry.GridMap ((!))
import Math.Geometry.GridMap.Lazy (lazyGridMap)

main = do
    let putStrLn' str = putStrLn ('\n':str)
    putStrLn' "Indices of hex grid:"
    print $ indices hex
    putStrLn' "Neighbors around (1,1) of hex grid:"
    print $ neighbours hex (1,1)
    putStrLn' "Indices of rect grid:"
    print $ indices rect
    putStrLn' "Neighbors around (1,1) of rect grid:"
    print $ neighbours rect (1,1)
    putStrLn' "value of hex at index (1,1)"
    print $ hexM ! (1,1)

hex = hexHexGrid 4

rect = rectSquareGrid 3 5

hexM = lazyGridMap hex [1..]
```

Rysunek 24: Kod sieci sześciokątne i kwadratowe.

```
rafcio@rafcio-M-Linux: ~/Pulpit/Haskell_projects/4. Hexa
Plik  Edycja  Widok  Wyszukiwanie  Terminal  Pomoc
rafcio@rafcio-M-Linux:~/Pulpit/Haskell_projects/4. Hexa$ ./hexa.x

Indices of hex grid:
[(-3,0),(-3,1),(-3,2),(-3,3),(-2,-1),(-2,0),(-2,1),(-2,2),(-2,3),(-1,-2),(-1,-1)
,(-1,0),(-1,1),(-1,2),(-1,3),(0,-3),(0,-2),(0,-1),(0,0),(0,1),(0,2),(0,3),(1,-3)
,(1,-2),(1,-1),(1,0),(1,1),(1,2),(2,-3),(2,-2),(2,-1),(2,0),(2,1),(3,-3),(3,-2),
(3,-1),(3,0)]

Neighbors around (1,1) of hex grid:
[(0,1),(0,2),(1,2),(2,1),(2,0),(1,0)]

Indices of rect grid:
[(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2),(3,0),(3,1),(3,2),(4,0),(
4,1),(4,2)]

Neighbors around (1,1) of rect grid:
[(1,2),(1,0),(2,1),(0,1)]

value of hex at index (1,1)
27
rafcio@rafcio-M-Linux:~/Pulpit/Haskell_projects/4. Hexa$
```

Rysunek 25: Rezultat sieci kwadratowe i prostokątne.

Literatura

[1] https://pl.wikipedia.org/wiki/Wierzcho

[2] https://en.wikipedia.org/wiki/Adjacency$_l ist$

[3] https://pl.wikipedia.org/wiki/Sortowanie$_t opologiczne$

[4] https://en.wikipedia.org/wiki/Dependency$_n etwork$

[5] https://en.wikipedia.org/wiki/Dependency$_n etwork$

[6] https://pl.wikipedia.org/wiki/Big$_d ataChapter$" *Graph Fundamentals*"