Pakiet `igraph` w R

Algorytmy Grafowe

Joanna Raczek
joanna.raczek@pg.edu.pl
Politechnika Gdańska

Spis treści

Rozdział 1

Wprowadzenie

`igraph` jest biblioteką fukcji języka R służącą do analizy grafów i sieci. Głównymi zadaniami realizowanymi przez `igraph` są:

- umożliwienie bezproblemowego korzystania z algorytmów grafowych,
- szybkie przetwarzanie dużych grafów o milionach wierzchołków i krawędziach,
- umożliwienie połączenia pakietu z językiem programowania wysokiego poziomu, jakim jest R.

Niniejsze krótkie opracowanie powstało na kanwie laboratorium z przedmiotu Algorytmy Grafowe prowadzonego na wydziale Fizyki Technicznej i Matematyki Stosowanej Politechniki Gdańskiej i ma na celu ułatwienie studentom rozpoczęcia pracy z biblioteką `igraph`.

Głównym materiałem źródłowym, na którym bazowano w trakcie powstawaniu tego opracowania, jest dokumentacja dostępna znajdująca się tu: https://igraph.org/r/

1.1 Czym jest graf?

Pakiet `igraph` umożliwia tworzenie grafów, czyli obiektów typu `IGRAPH`. Przykładowo, poniższym poleceniem tworzymy cykl nieskierowany o 10 wierzchołkach.

```
g <- make_ring(10, directed = FALSE)
```

W zmiennej g przechowywany jest obiekt typu IGRAPH o następujących własnościach:

```
IGRAPH U--- 10 10 -- Ring graph
+ attr: name (g/c), mutual (g/l), circular (g/l)
+ edges:
 [1] 1-- 2 2-- 3 3-- 4 4-- 5 5-- 6 6-- 7 7-- 8 8-- 9
     9--10 1--10
```

Litera U oznacza, że graf jest nieskierowany. Na tym samym miejscu może być litera D w przypadku grafu skierowanego. Na drugim miejscu może pojawić się litera N jeśli wierzchołki grafu mają przyporządkowane nazwy. Litera W na trzecim miejscu oznacza graf ważony (z wagami na krawędziach), a litera B na czwartym miejscu pojawia się w przypadku grafu dwudzielnego. Nie jest atrybut ustawiany automatycznie — cykl C_{10} jest grafem dwudzielnym, ale nie ma ustawionej tej flagi. Kolejne dwie liczby to liczba wierzchołków i liczba krawędzi grafu. Po dwóch minusach mamy nazwę grafu (o ile jest ustawiona). W naszym przykładzie nazwa to `Ring graph`.

Druga linia jest opcjonalna; zawiera informacje na temat atrybutów grafu. Graf w naszym przykładzie ma ustawiony atrybut `name` typu `character` oraz atrybuty `mutual` i `circular` typu złożonego. Typ złożony w R to każdy inny typ oprócz typu liczbowego lub znakowego.

Kolejna linia wyświetla krawędzie grafu.

Przykład 1: Drzewo skierowane

Dla porównania, przyjrzyjmy się obiektowi klasy IGRAPH, który przechowuje drzewo skierowane.

```
t <- make_tree(15, 3)

IGRAPH D--- 15 14 -- Tree
+ attr: name (g/c), children (g/n), mode (g/c)
+ edges:
```

```
 [1] 1-> 2 1-> 3 1-> 4 2-> 5 2-> 6 2-> 7 3-> 8 3-> 9
     3->10 4->11 4->12 4->13 5->14 5->15
```

Przykład 2: Graf losowy dwudzielny

Graf dwudzielny skierowany.

```
gb <- sample_bipartite(3, 8, p=0.6, directed=TRUE, mode="in")

IGRAPH D--B 11 14 -- Bipartite Gnp random graph
+ attr: name (g/c), p (g/n), type (v/l)
+ edges:
 [1]  4->1  4->3  5->2  5->3  6->2  7->3  8->1  8->2
      9->1  9->2  9->3 10->1 10->3 11->2
```

Powyższy przykład tworzy losowy graf dwudzielny, więc za każdym razem wynik może być nieco inny. Poniżej rysunek 1.1 przedstawia przykład takiego grafu. Grafy wygodnie rysuje się za pomocą polecenia `tkplot`.

```
tkplot(gb, vertex.color="green", edge.width=3)
```

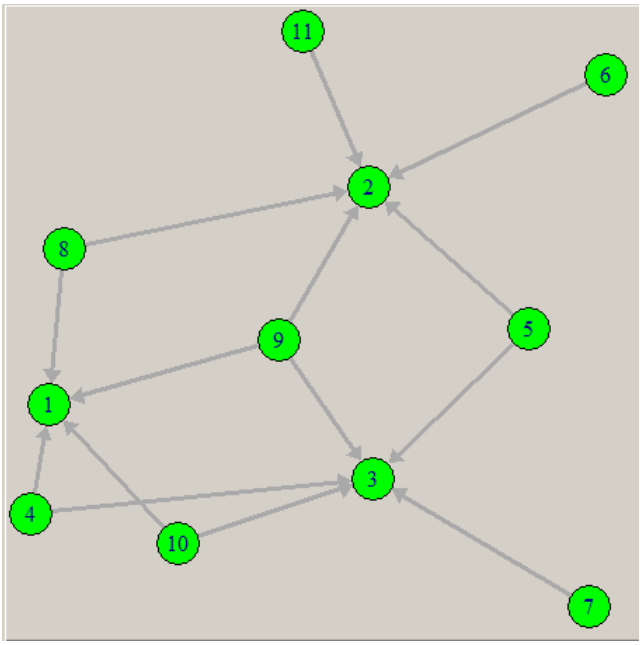
Dodatkowo, ustawiliśmy kolor wierzchołków na zielono, a grubość krawędzi na 3.

1.2 Tworzenie grafów

Pakiet `igraph` dostarcza kilka możliwości tworzenia grafów, zarówno deterministycznych jak i losowych.

1.2.1 Funkcjia `make_graph`

Na początek przyjrzymy się funkcji `make_graph`. Składnia funkcji ma następującą postać



Rysunek 1.1: Graf dwudzielny skierowany

```
make_graph(edges, ..., n = max(edges), isolates = NULL,
           directed = TRUE, dir = directed, simplify = TRUE)
```

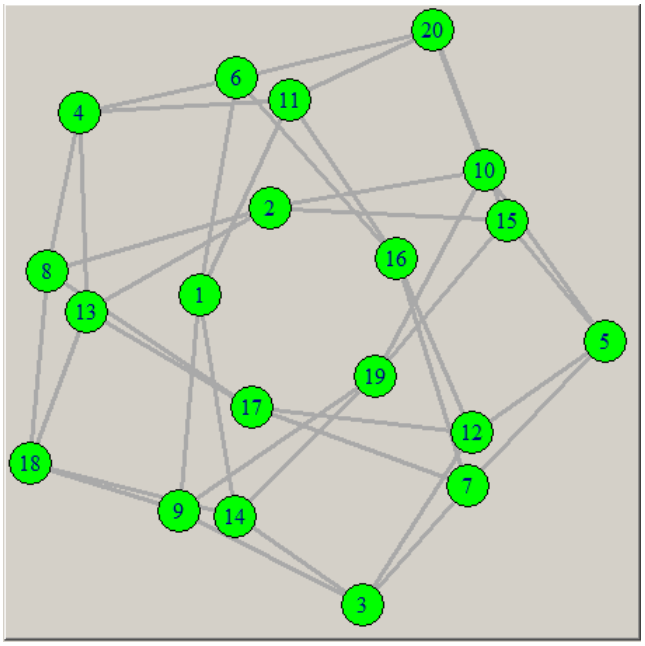
Przykład 3: Graf Folkmana

Aby utworzyć graf Folkmana należy napisać

```
make_graph("Folkman")

IGRAPH U--- 20 40 -- Folkman
+ attr: name (g/c)
+ edges:
 [1]  1-- 6  1-- 9  1--11  1--14  2-- 8  2--10  2--13
      2--15  3-- 7  3-- 9  3--12
[12]  3--14  4-- 6  4-- 8  4--11  4--13  5-- 7  5--10
      5--12  5--15  6--16  6--20
```

```
[23]  7--16  7--17  8--17  8--18  9--18  9--19 10--19
     10--20 11--16 11--20 12--16
[34] 12--17 13--17 13--18 14--18 14--19 15--19 15--20
```

Graf Folkmana jest zilustrowany na rysunku 1.2



Rysunek 1.2: Graf Folkmana

1.2.2 Funkcja `graph_from_literal`

Funkcja ta nadaje się do tworzenia niewielkich grafów.

```
graph_from_literal(..., simplify = TRUE)
```

Pierwszy argument funkcji to podane w bezpośredni sposób krawędzie grafu. Drugi argument domyślnie przyjmuje wartość logiczną `TRUE`, co spowoduje, że podczas tworzenia grafu zostaną usunięte krawędzie wielokrotne oraz pętle.

Przykład 4: Wykorzystanie funkcji `graph_from_literal`

Graf o czterech niezależnych krawędziach i trzech wierzchołkach izolowanych, zilustrowany na rysunku 1.3:

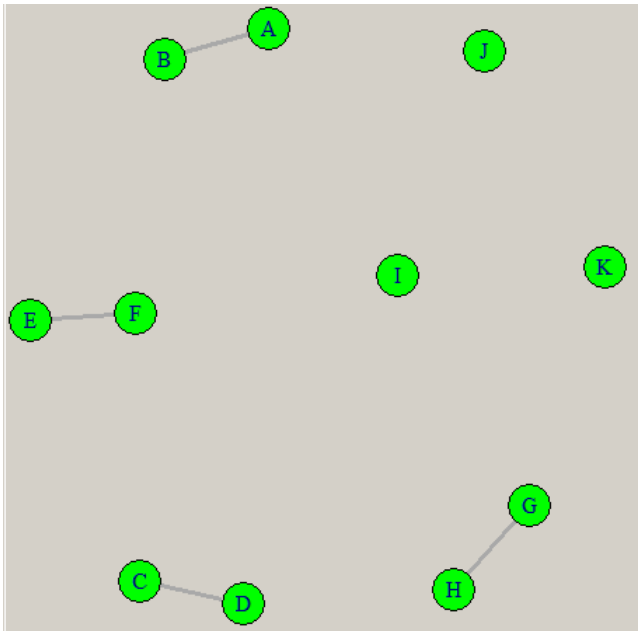
```
graph_from_literal( A--B, C--D, E--F, G--H, I, J, K )
```

Graf pełny, prosty:

```
graph_from_literal( A:B:C:D -- A:B:C:D )
```

Graf skierowany:

```
graph_from_literal( A +- B -+ C )
```



Rysunek 1.3: Graf z przykładu 4

1.2.3 Funkcja `graph_from_edgelist`

1.2.4 Funkcja `graph_from_adjacency_matrix`

Funkcja `graph_from_adjacency_matrix` tworzy graf na podstawie macierzy sąsiedztwa. Zilustrujemy na przykładzie tworzenie grafu prostego, ważonego na podstawie danej macierzy sąsiedztwa.

Przykład 5: Graf na podstawie macierzy sąsiedztwa

Na początku tworzymy macierz podając jej wyrazy. Może ona być wczytana z pliku.

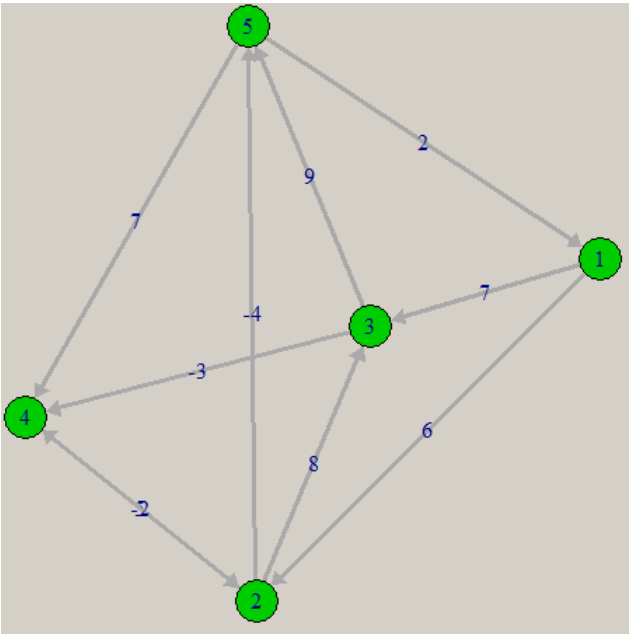
```
M=matrix(0,nrow=5, ncol=5)
M[1,2]=6
M[1,3]=7
M[2,4]=5
M[2,5]=-4
M[2,3]=8
M[3,5]=9
M[3,4]=-3
M[4,2]=-2
M[5,4]=7
M[5,1]=2
```

```
> M
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    6    7    0    0
[2,]    0    0    8    5   -4
[3,]    0    0    0   -3    9
[4,]    0   -2    0    0    0
[5,]    2    0    0    7    0
```

Następnie tworzymy graf i rysujemy go (patrz rys. 1.4.

```
g2 <- graph_from_adjacency_matrix(M, mode = c("directed"),
      weighted=TRUE)
tkplot(g2, edge.label=E(g2)$weight, vertex.color=3,
```

```
        edge.width=3)
```



Rysunek 1.4: Graf z przykładu 5

1.2.5 Funkcja `graph_from_incidence_matrix`

1.2.6 Funkcja `make_full_graph`

I inne jej podobne, jak tworzenie grafu dwudzielnego pełnego.

Rozdział 2

Własności macierzy sąsiedztwa

Analizując macierz sąsiedztwa grafu prostego, nieważonego, można określić wiele własności grafu. W rozdziale tym zajmiemy się dwoma z nich: ilością trójkątów w grafie oraz ilością drzew spinających grafu.

2.1 Trójkąty w grafie

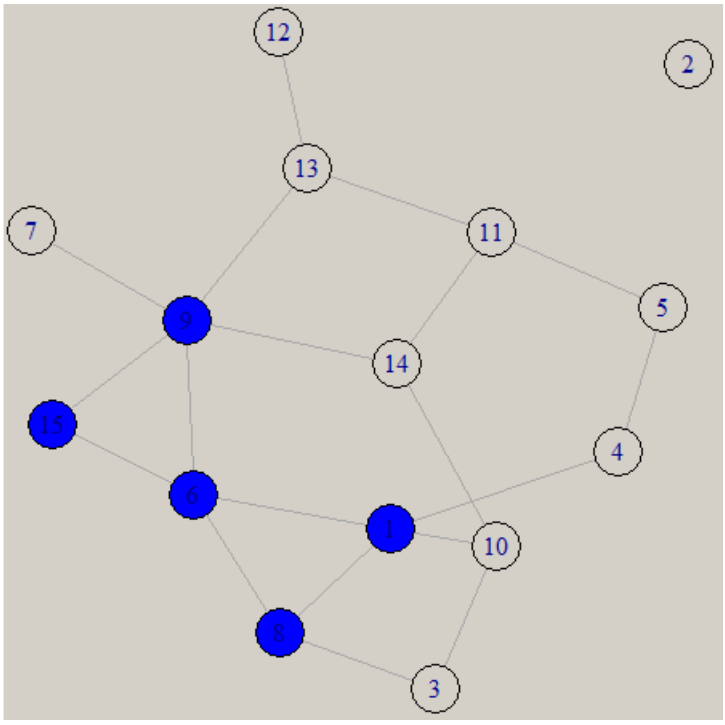
Jeśli A jest macierzą sąsiedztwa grafu prostego, to element $[i, j]$ n-tej potęgi macierzy A, czyli $A^n[i, j]$, jest liczbą spacerów o długości n (czyli o n krawędziach) między wierzchołkami i oraz j. Korzystając z tego, można obliczyć, ile trójkątów zawiera graf. Wystarczy macierz sąsiedztwa podnieść do potęgi trzeciej, dodać elementy na głównej przekątnej i podzielić przez 6 (gdyż każdy trójkąt jest w ten sposób zliczony 6 razy). Powyższy sposób postępowania reprezentuje algorytm `trojkaty`. Daną wejściową dla algorytmu jest graf.

```
g <- sample_gnp(15, 1/4)

trojkaty <- function(g){
  A <- as_adjacency_matrix(g,sparse=FALSE)
  B <- A %*% A %*% A
  sum(diag(B))/6
}
```

Program ten można rozbudować o kolorowanie wierzchołków wchodzących w skład jednego z trójkątów w następujący sposób.

```
trojkaty <- function(g){
A <- as_adjacency_matrix(g,sparse=FALSE)
  B <- A %*% A %*% A
  sum(diag(B))/6
  for (i in 1:nrow(A)){
    if (B[i,i] != 0)
      V(g)[i]$color <- 4
  }
  tkplot(g)
}
```

Wynikiem działanie algorytmu może być rysunek 2.1, gdzie wierzchołki wchodzące w skład dowolnego cyklu C_3 są niebieskie, a pozostałe wierzchołki są białe.



Rysunek 2.1: Graf z dwoma trójkątami

2.2 Liczba drzew spinających grafu prostego

Korzystając z twierdzenia Kirchhoffa można wyznaczyć ilość drzew spinających na podstawie analizy macierzy sąsiedztwa grafu prostego.

Twierdzenie 1: Kirchhoffa

Niech M będzie macierzą powstałą z macierzy sąsiedztwa pewnego spójnego grafu G poprzez zamianę każdej liczby 1 w tej macierzy na liczbę -1 oraz wpisaniu na głównej przekątnej stopni odpowiadających wierzchołków. Wtedy liczba drzew spinających grafu G jest równa wartości dowolnego dopełnienia algebraicznego (ang. *cofactor*) macierzy M.

Poniższy kod w R wyznacza ilość drzew spinających grafu podanego jako argument wejściowy według algorytmu opisanego w twierdzeniu Kirchhoffa.

```
drzewaSpinajace <- function(g){
  A <- as_adjacency_matrix(g,sparse=FALSE)
  B <- (-1)* A
  for (i in 1:nrow(A)){
    B[i,i] <- sum(A[i,])
    }
 det(B[-1,-1])
}
```

Inne rozwiązanie poniżej. Obiekt G jest grafem, danym do funkcji.

```
kirchoff <- function(G){
  M = as.matrix(as_adj(G))
  M[which(M==1)] = -1
  diag(M) = degree(G)
  return(det(M[-1,-1]))
}
```

Rozdział 3

Najkrótsze ścieżki w grafie

Pakiet R oferuje kilka funkcji znajdujących najkrótsze ścieżki w grafie. Wykorzystują one algorytm Dijkstry, Johnsona, Bellmana – Forda lub BFS (przeszukiwanie grafu wszerz) w przypaku grafów bez wag na krawędziach.

3.1 Algorytm Dijkstry

Edsger Dijkstra był holenderskim fizykiem, matematykiem i informatykiem. Żył w latach 1930–2002. Najbardziej znany jest dzięki algorytmowi znajdowania najkrótszych ścieżek w grafie oraz problemowi pięciu ucztujących filozofów. Otrzymał nagrodę Turinga za wkład do algorytmiki i języków programowania.

Algorytm Dijkstry pozwala na znalezienie najkrótszej drogi z pewnego wybranego wierzchołka do innego wybranego wierzchołka w grafie ważonym o nieujemnych wagach krawędzi. **Graf ważony** to graf, w którym każdej krawędzi e przyporządkowano liczbę rzeczywistą $w(e)$, nazywaną wagą krawędzi. **Wagą grafu (podgrafu)** nazywamy sumaryczną wartość wszystkich wag krawędzi grafu (podgrafu). Przedstawiony algorytm 1 znajduje najkrótszą drogę od wierzchołka A do E przy okazji znajdując drogę do innych wierzchołków grafu.

Algorytm 1: Algorytm Dijkstry

Krok 1 Przypisz wierzchołkowi A stałą etykietę $(-, 0)$, a pozostałym wierzchołkom tymczasową etykietę $(-, \infty)$.

Krok 2 Dopóki E nie ma przypisanej etykiety stałej i istnieją wierzchołki, którym można przypisać etykietę stałą, rób:

(a) Niech u będzie wierzchołkiem, który jako ostatni otrzymał etykietę stałą $u(x, d)$. Dla każdego wierzchołka v bez stałej etykiety i sąsiedniego do u oblicz $d' = d + w(uv)$. Jeśli d' jest mniejsze niż aktualna etykieta v, ustaw $v(u, d')$.

(b) Spośród wszystkich wierzchołków bez stałej etykiety wybierz ten o najmniejszej etykiecie i ustaw etykietę jako stałą.

Złożonośc obliczeniowa przedstawionego tu algorytmu Dijkstry wynosi $O(n^2)$, gdzie n jest ilością wierzchołków grafu.

3.2 Funkcja `shortest.paths`

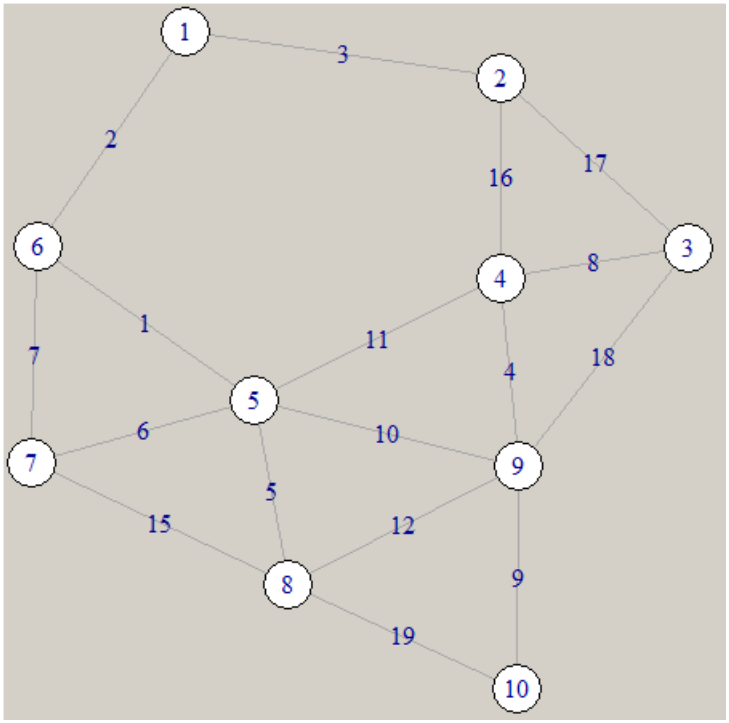
Przedstawimy działanie funkcji `shortest.paths` na przykładzie grafu G danego macierzą sąsiedztwa M.

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
 [1,]    0    3    0    0    0    2    0    0    0     0
 [2,]    3    0   17   16    0    0    0    0    0     0
 [3,]    0   17    0    8    0    0    0    0   18     0
 [4,]    0   16    8    0   11    0    0    0    4     0
 [5,]    0    0    0   11    0    1    6    5   10     0
 [6,]    2    0    0    0    1    0    7    0    0     0
 [7,]    0    0    0    0    6    7    0   15    0     0
 [8,]    0    0    0    0    5    0   15    0   12    13
 [9,]    0    0   18    4   10    0    0   12    0     9
[10,]    0    0    0    0    0    0    0   19    9     0
```

Na podstawie macierzy sąsiedztwa M tworzymy w R graf nieskierowany ważony oraz rysujemy go.

```
G <- graph_from_adjacency_matrix(M, weighted=TRUE,
     mode = c("undirected"))
tkplot(G,edge.label=E(G)$weight,vertex.color="white")
```

Wynikiem poleceń jest graf G zilustrowany na rysunku 3.1.



Rysunek 3.1: Graf G

Przykład 6: Algorytm Dijkstry

Korzystając z funkcji `shortest.paths`, w której wybieramy algorytm Dijkstry,

```
shortest.paths(G, v=1, to=V(G), algorithm="dijkstra")
```

uzyskujemy odległości od wierzchołka 1 do każdego innego wierzchołka grafu za pomocą algorytmu Dijkstry.

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    0    3   20   14    3    2    9    8   13    22
```

Podobnie możemy znaleźć odległości pomiędzy każdą parą wierzchołków tego grafu.

```
shortest.paths(G, v=V(G), to=V(G), algorithm="dijkstra")

      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
 [1,]    0    3   20   14    3    2    9    8   13    22
 [2,]    3    0   17   16    6    5   12   11   16    25
 [3,]   20   17    0    8   19   20   25   24   12    21
 [4,]   14   16    8    0   11   12   17   16    4    13
 [5,]    3    6   19   11    0    1    6    5   10    19
 [6,]    2    5   20   12    1    0    7    6   11    20
 [7,]    9   12   25   17    6    7    0   11   16    25
 [8,]    8   11   24   16    5    6   11    0   12    19
 [9,]   13   16   12    4   10   11   16   12    0     9
[10,]   22   25   21   13   19   20   25   19    9     0
```

3.3 Funkcja `shortest_paths`

Korzystając z możliwości funkci `shortest_paths` możemy uzyskać więcej informacji o znalezionych najkrótszych ścieżkach. Korzystając z grafu takiego jak w poprzednij sekcji, możemy wyświetlić wierzchołki wchodzące w skład najkrótszej ścieżki.

```
shortest_paths(g2, 1,to=V(g2))

$vpath
$vpath[[1]]
+ 0/10 vertices:

$vpath[[2]]
+ 2/10 vertices:
[1] 1 2

$vpath[[3]]
+ 3/10 vertices:
[1] 1 2 3

$vpath[[4]]
```

```
+ 4/10 vertices:
[1] 1 6 5 4

$vpath[[5]]
+ 3/10 vertices:
[1] 1 6 5

$vpath[[6]]
+ 2/10 vertices:
[1] 1 6

$vpath[[7]]
+ 3/10 vertices:
[1] 1 6 7

$vpath[[8]]
+ 4/10 vertices:
[1] 1 6 5 8

$vpath[[9]]
+ 4/10 vertices:
[1] 1 6 5 9

$vpath[[10]]
+ 5/10 vertices:
[1]  1  6  5  9 10
```

Pokolorujemy wierzchołki należące do najkrótszej ścieżki prowadzącej od wierzchołka 1 do 10.

Przykład 7: Kolorowanie wierzchołków z najkrótszej ścieżki

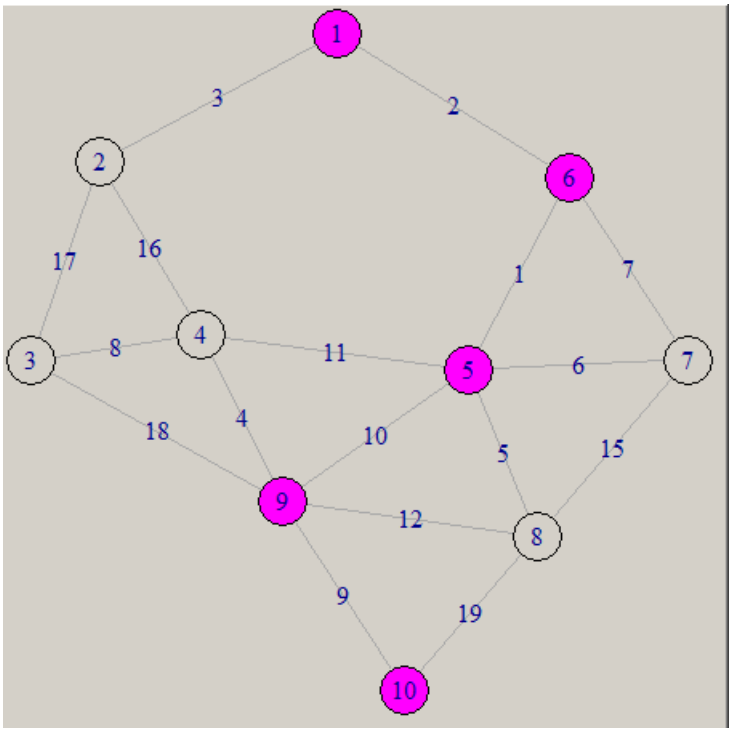
Zapisujemy wierzchołki ze ścieżki, która nas interesuje, do zmiennej.

```
sciezka <- shortest_paths(G, 1,to=V(G))$vpath[[10]]
```

Nadajemy wierzchołkom kolor i rysujemy graf.

```
V(G)[sciezka]$color <- 6
tkplot(G,edge.label=E(G)$weight)
```

W wyniku powyższych poleceń otrzymujemy graf jak na rysunku 3.2



Rysunek 3.2: Graf G z zaznaczoną najkrótszą ścieżką

Dodatkowo możemy oznaczyć krawędzie tej najkrótszej ścieżki. Opcja `epath` spowoduje wyświetlenie krawędzi należących do najkrótszych ścieżek. Ponownie skupimy się na najkrótszej ścieżce wiodącej od wierzchołka 1 do 10.

Przykład 8: Kolorowanie krawędzi najkrótszej ścieżki

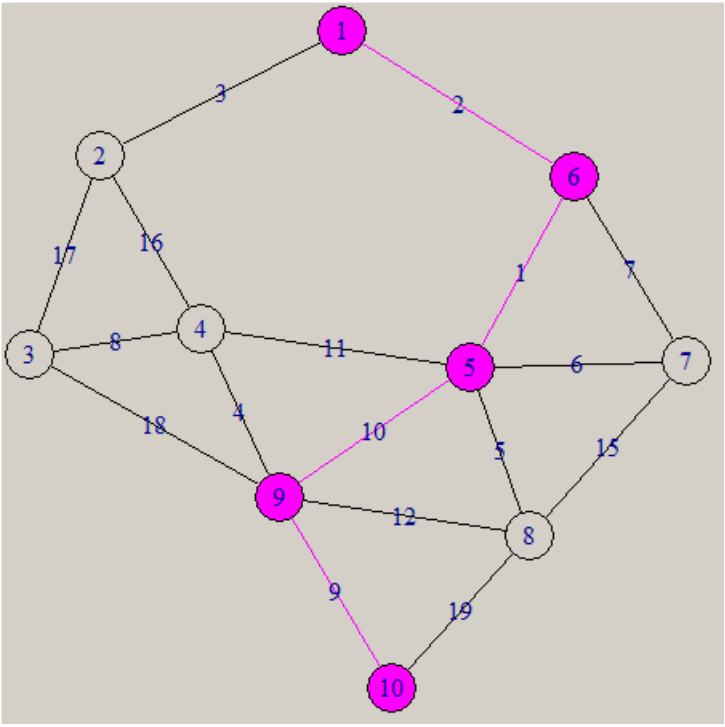
Zapisujemy krawędzie ścieżki, która nas interesuje, do zmiennej.

```
sciezka2 <- shortest_paths(G, 1, to=V(G),
            output = c("epath"))$epath[[10]]
```

Nadajemy wszystkim krawędziom kolor 1, a pozostałym kolor 6. Rysujemy graf.

```
E(G)$color <- 1
E(G)[sciezka2]$color <- 6
tkplot(G,edge.label=E(G)$weight)
```

W wyniku powyższych poleceń otrzymujemy graf jak na rysunku 3.3.



Rysunek 3.3: Graf G z zaznaczoną najkrótszą ścieżką

3.4 Algorytm Bellmana–Forda

Richard Ernest Bellman żył w latach 1920–1984. Urodził się w Nowym Jorku. Był synem niepraktykujących Żydów pochodzenia polskiego i rosyjskiego. Jego ojciec prowadził mały warzywniak. Studiował matematykę. Pracował jako programista. Wynalazł programowanie dynamiczne w 1953 roku. Interesował

się też równaniami różniczkowymi cząstkowymi. Swój algorytm znajdujący najktósze ścieżki w grafach z krawędziami o ujemnych wagach opublikował w 1958 roku.

Lester Randolph Ford, Jr. urodził się w 1927 roku w Houston. Jest synem matematyka Lestera R. Forda, Sr. Swój algorytm znajdujący najktósze ścieżki w grafach z krawędziami o ujamnych wagach opublikował w 1956 roku. W tym samym czasie razem z Fulkersonem zajmował się sieciami przepływowymi. Ci panowie są autorami twierdzenia o maksymalnym przepływie i minimalnym przekroju w sieciach przepływowych.

Dodatkowo w 1957 roku algorytm znajdujący najkrótsze ścieżki w grafach z krawędziami o ujemnych wagach opublikował Edward E. Moore, przez co czasami ten algorytm nazywany jest algorytmem Bellmana–Forda–Moore'a.

Dany jest skierowany graf ważony, w którym waga każdej krawędzi jest liczbą dodatnią lub ujemna lub zerem. Chcemy znać długość najkrótszych dróg pomiędzy ustalonym wierzchołkiem v_1 a każdym innym wierzchołkiem grafu lub w przypadku istnienia cyklu ujemnego, informację o tym, że najkrótszej drogi nie ma.

Algorytm 2: Algorytm Bellmana–Forda

Krok 1 Dla $i, j = 1, 2, \ldots, n$ zrób:

- jeśli $i = j$ niech $w(i, j) = 0$;
- jeśli $i \neq j$ i
  - jeśli $v_i v_j$ jest łukiem, niech $w(i, j) = w(v_i v_j)$,
  - jeśli $v_i v_j$ nie jest łukiem, niech $w(i, j) = \infty$;
- ustaw $d_0(1) = 0$ oraz $d_0(j) = \infty$ dla $j = 2, 3, \ldots, n$;
- ustaw $p(i) = 1$ dla $i = 2, 3, \ldots, n$.

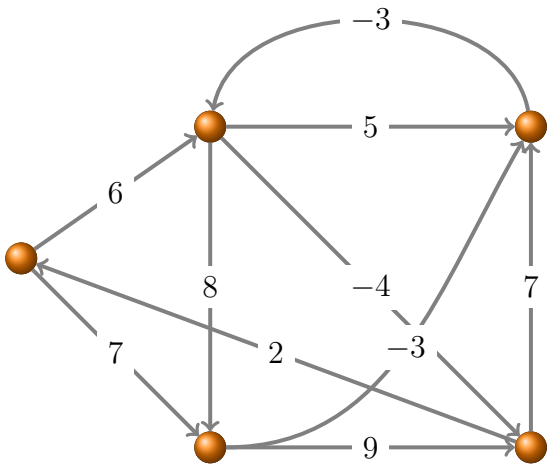
Krok 2 Dla $i = 1, 2, \ldots, n$ zrób:

- dla $j = 1, 2, \ldots, n$ zrób:
  - znajdź liczbę k dla której wyrażenie $\min = d_{i-1}(k) + w(k, j)$ jest najmniejsze,
  - jeśli $\min < d_{i-1}(j)$
    ustaw $d_i(j) = \min$ i ustaw $p(j) = v_k$,

a w przeciwnym wypadku
ustaw $d_i(j) = d_{i-1}(j)$;

Krok 3 Jeśli $d_n(j) = d_{n-1}(j)$ dla każdego $j = 1, 2, \ldots, n$,
to wynikiem są liczby $d_n(1), d_n(2), \ldots, d_n(n)$,
w przeciwnym wypadku wypisz
„Nie ma najkrótszych ścieżek. Jest cykl ujemny."

Zilustrujemy działanie algorytmu i wykorzystanie R dla grafu zilustrowanego na rys. 3.4.



Rysunek 3.4: Graf skierowany z wagami ujemnymi

Macierz sąsiedztwa tego grafu jest następująca:

```
      [,1] [,2] [,3] [,4] [,5]
[1,]    0    6    7    0    0
[2,]    0    0    8    5   -4
[3,]    0    0    0   -3    9
[4,]    0   -3    0    0    0
[5,]    2    0    0    7    0
```

Na podstawie macierzy sąsiedztwa M tworzymy w R graf skierowany ważony.

```
G <- graph_from_adjacency_matrix(M, weighted=TRUE)
```

Korzystając z algorytmu Bellmana–Forda znajdujemy najkrótsze ścieżki od wierzchołka 1 do każdego innego.

```
shortest.paths(G, 1, to=, algorithm="bellman-ford")

Error in .Call("R_igraph_shortest_paths", graph, v - 1,
      to - 1, as.numeric(mode),  :
      At structural_properties.c:5235 : cannot run
      Bellman-Ford algorithm,
      Negative loop detected while calculating
      shortest paths
```

Pojawił się błąd — w grafie istnieje cykl ujemny, nie ma najkrótszych ścieżek. Gdyby najkrótsza ścieżka istniała, to chodząc wzdłuż cyklu ujemnego moglibyśmy dowolnie dowolnie ją skrócić, czyli znaleźlibyśmy krótszą niż najkrótszą, sprzeczność.

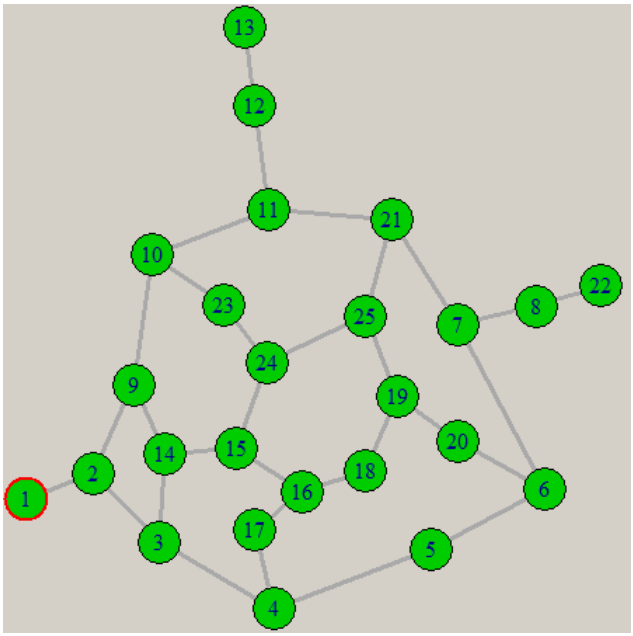
Rozdział 4

Przeszukiwanie grafu wszerz i w głąb

4.1 Przeszukiwanie w głąb

Zilustrujemy przeszukiwanie grafu w głąb na przykładzie grafu Waltera zilustrowanego na rysunku 4.1. Korzystamy z funkcji `graph.dfs`, której wynik działania jest następujący.

```
pom2 <- graph.dfs(g, root = 1, father = TRUE)

$root
[1] 0

$neimode
[1] "out"

$order
+ 25/25 vertices:
 [1]  1  2  3  4  5  6  7  8 22 21 11 10  9 14 15 16 17 18
     19 20 25 24 23 12 13

$order.out
NULL
```



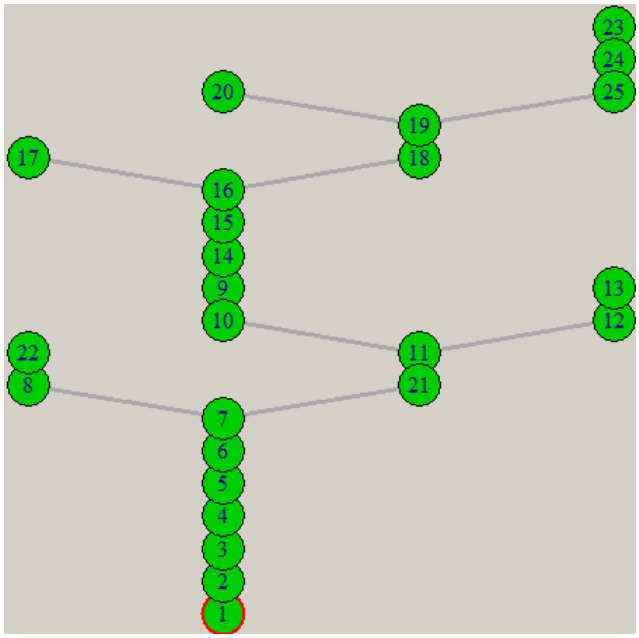
Rysunek 4.1: Graf Waltera

```
$father
+ 25/25 vertices:
 [1] NA  1  2  3  4  5  6  7 10 11 21 11 12  9 14 15 16 16
     18 19  7  8 24 25 19

$dist
NULL
```

Wynik działania funkcji nie jest zbyt czytelny, ale w łatwy sposób można sobie z tym poradzić. Na początek tworzymy drzewo przeszukiwania i wyświetlamy je (rys. 4.2).

```
h <- graph(rbind(pom2$order,
     pom2$father[pom2$order, na_ok = TRUE])[,-1],
     directed=FALSE )

tkplot(h, vertex.color=3, edge.width=3)
```

Następnie tworzymy tablicę krawędzi wykorzystanych w przeszukiwaniu



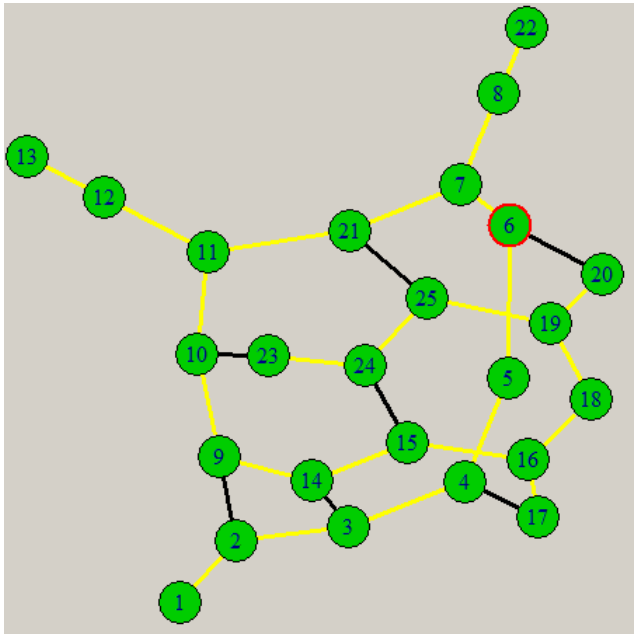
Rysunek 4.2: Drzewo przeszukiwania w głąb grafu Waltera

w głąb, zaznaczamy je na żółto i rysujemy graf. Jednak na samym początku warto wszystkie wierzchołki i krawędzie pokolorować na inne, niż żółty, kolory.

```
V(g)$color <- 3
E(g)$color <-1
E(g)$width <- 3

# tablica krawędzi
pp <- rbind(pom2$order,
      pom2$father[pom2$order, na_ok = TRUE])[,-1]

# id znalezionych krawędzi
ppp <- get.edge.ids(g, pp)

# kolorowanie ich na zółto
E(g)$color[ppp] <- 7
```

```
tkplot(g)
```

Wynik działania ilustruje rys. 4.3.



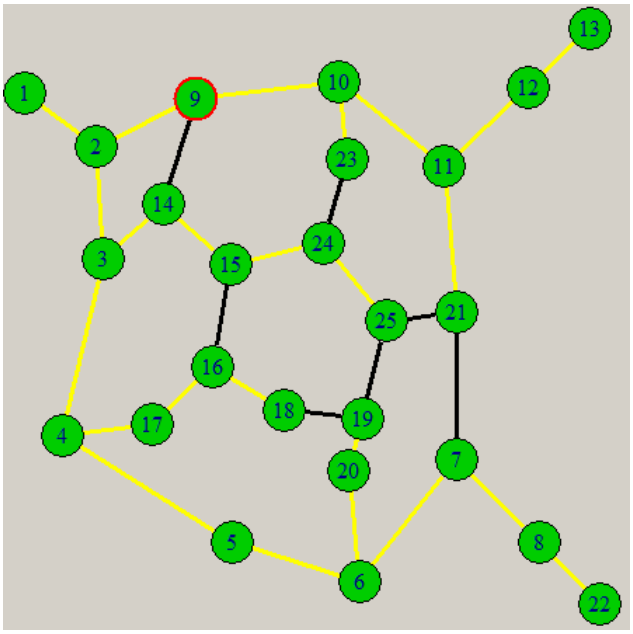
Rysunek 4.3: Krawędzie drzewa przeszukiwania w głąb zaznaczone w grafie Waltera

4.2 Przeszukiwanie wszerz

Podobnie.

```
# ojcowie do krawedzi
pom2 <- graph.bfs(g, root = 1, father = TRUE)

V(g)$color <- 3
E(g)$color <-1
E(g)$width <- 3

#drzewo przeszukiwania
h <- graph(rbind(pom2$order,
```

```
     pom2$father[pom2$order, na_ok = TRUE])[,-1],
     directed=FALSE)
tkplot(h, vertex.color=3, edge.width=3)

# tabela krawędzi
pp <- rbind(pom2$order,
      pom2$father[pom2$order, na_ok = TRUE])[,-1]

# id znalezionych krawędzi
ppp <- get.edge.ids(g, pp)
# kolorowanie ich na zółto
E(g)$color[ppp] <- 7

tkplot(g)
```



Rysunek 4.4: Krawędzie drzewa przeszukiwania wszerz zaznaczone w grafie Waltera

Rozdział 5

Kolorowanie grafów w R

Optymalne pokolorowanie grafu jest problemem trudnym obliczeniowo.

5.1 Algorytm LF

Algorytm FL jest prostym algorytmem sekwencyjnym kolorującym grafy. Nazwa algorytmu pochodzi od angielskich słów *largest first*. Wymyślony został przez Welsha i Powella. Jego działanie opiera się na obserwacji, że wierzchołki małego stopnia najlepiej kolorować jak najpóźniej. Jego złożoność jest wielomianowa.

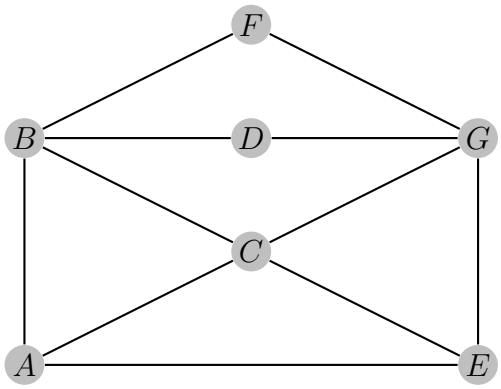
Algorytm 3: Algorytm LF

Krok 1 Posortuj wierzchołki według stopni nierosnąco.

Krok 2 Pokoloruj wierzchołki zachłannie (pierwszym wolnym kolorem) w kolejności wyznaczonej przez posortowanie.

Algorytm może działać bardzo źle i pokolorować graf za pomocą $\frac{n}{2}$ kolorów zamiast dwóch (np. grafy Johnsona J_k). Najmniejszym dość trudnym do pokolorowania grafem jest P_6. Najmniejszym trudnym do pokolorowania grafem jest koperta, zilustrowana na rys. 5.1. Kopertę można pokolorować trzema kolorami, natomiast algorytm LF zawsze użyje czterech kolorów.

Algorytm LF można zaimplementować w R w następujący sposób.



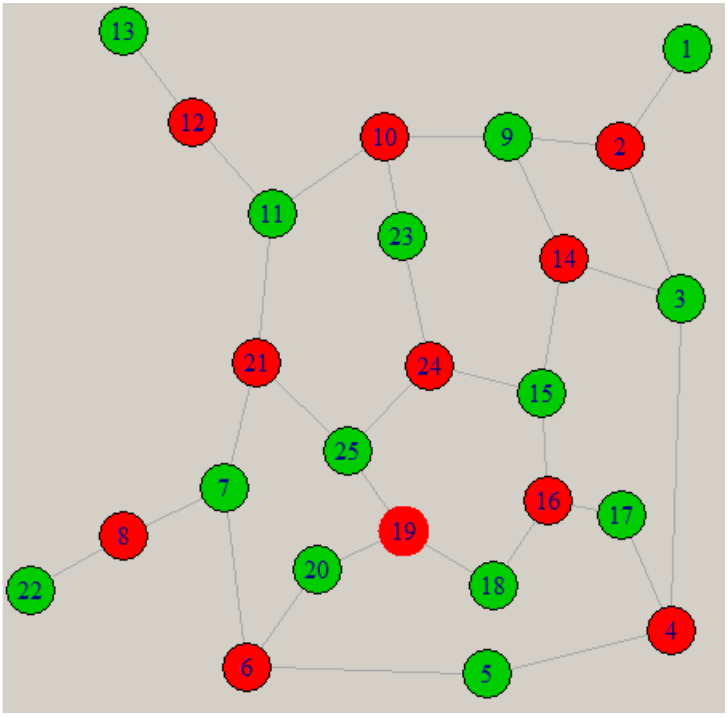
Rysunek 5.1: Koperta

```
LF <- function(A){
  g <- graph_from_adjacency_matrix(A, weighted = TRUE,
       "undirected")
  sg <- rbind(degree(g),1:length(degree(g)))
  sg <- sg[,order(degree(g), decreasing = TRUE)]

  k<-rep(0,times=nrow(A))
  for (i in 1:nrow(A)){
    kolor <- 1
    j<-0
   while (j < nrow(A)){
     j <- j+1
        if(A[sg[2,i],j]==1 & k[j]==kolor){
          kolor <- kolor+1
          j <- 0
        }
      }
    k[sg[2,i]] <- kolor
  }

  V(g)$color <- (k+1)
  tkplot(g, layout = layout_with_kk)
}
```

Zmienna tablicowa `sg` przechowuje stopnie wierzchołków (pierwszy wiersz tablicy `sg`) wraz z odpowiadającymi im numerami wierzchołków (drugi wiersz tablicy `sg`). Następnie tablica jest sortowana kolumnami według stopni wierzchołków nierosnąco.

W kolejnych krokach każdy wierzchołek jest kolorowany pierwszym wolnym kolorem. Kolorowanie odbywa się według kolejności wyznaczonej przez posortowanie tablicy `sg`. Kolory wierzchołków przechowuje wektor `k`. Przy poszukiwaniu jak najmniejszego wolnego koloru użyto pętli `while`, gdyż pętla `for` w R nie pozwala na manipulację w pętli zmienną indeksującą pętli. Przy kazdej zmianie tymczasowego koloru wierzchołka należy przeglądnać sąsiedztwo kolorowanego wierzchołka od początku. W przeciwnym wypadku możemy nie otrzymać prawidłowego pokolorowania grafu.

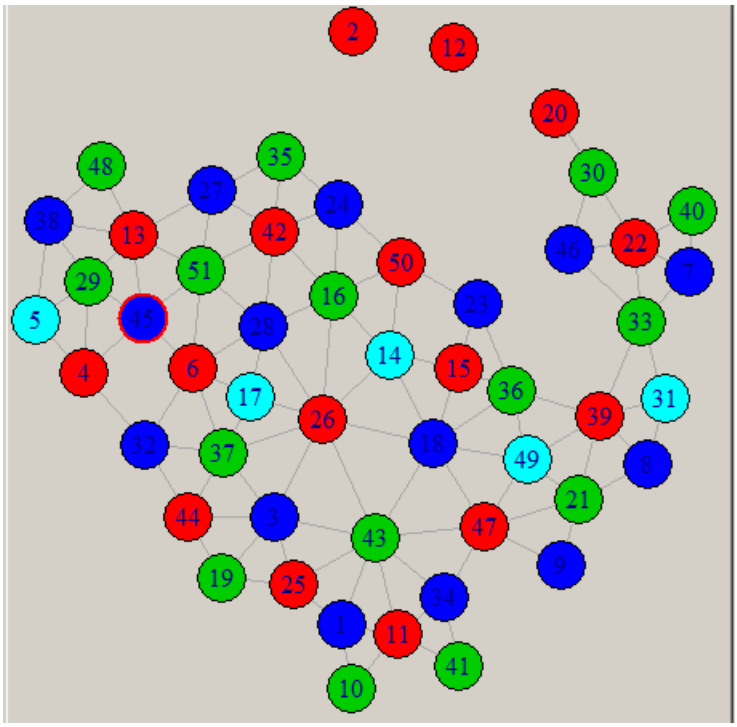
Na koniec kolory przepisywane są z tablicy `k` do wierzchołków i rysowany jest graf z pokolorowanymi wierzchołkami. Dodajemy 1 do numeru koloru aby uniknąć czarnych wierzchołków.



Rysunek 5.2: Graf Waltera pokolorowany algorytmem LF

Rysunek 5.2 ilustruje efekt działania funkcji dla grafu Waltera. Graf ten

został pokolorowany przez naszą implementację algorytmu LF w sposób optymalny. Z kolei graf wizualizujący strukturę stanów USA został pokolorowany czterema kolorami (rys. 5.3).



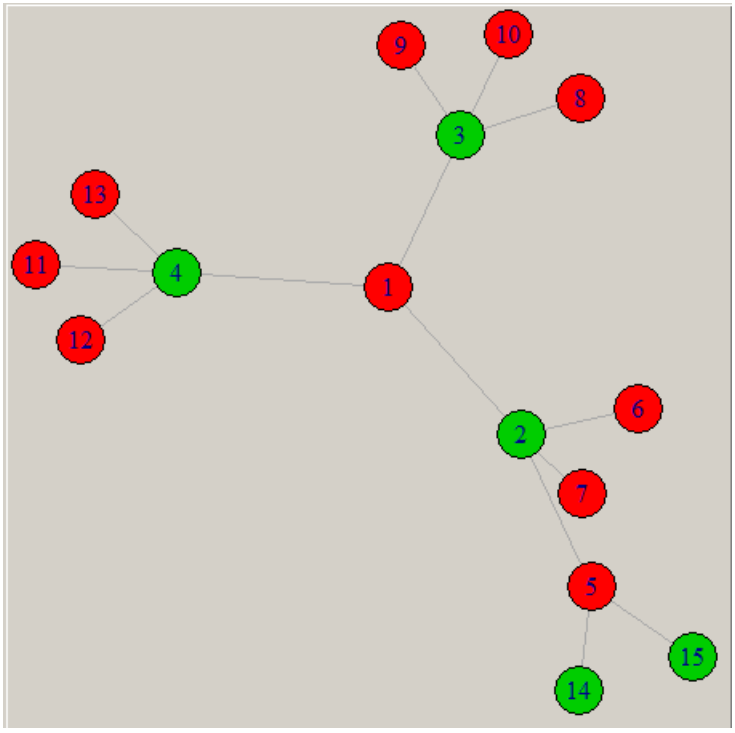
Rysunek 5.3: Graf USA pokolorowany algorytmem LF

5.2 Optymalne kolorowanie grafów dwudzielnych

Grafy dwudzielne można w czasie wielomianowym pokolorować dwoma kolorami, czyli pokolorować optymalnie. Do kolorowania grafów dwudzielnych spójnych użyjemy gotowej funkcji dostępnej w R, `distances`, znajdującej odległości między podanym wierzchołkiem v a dowolnym innym. Ponieważ graf jest dwudzielny, żadne dwa wierzchołki o parzystej odległości od v nie są sąsiednie; podobnie nie są sąsiednie żadne dwa wierzchołki o nieparzystej odległości od v. Kolor wierzchołka jest więc równy reszcie odległości wierzchołka przy dzieleniu przez 2, (dodać 2 aby uniknąć czarnych wierzchołków).

```
t <- make_tree(15, 3, mode = "undirected")
kol <- distances(t, v=1)
kol <- (kol %%2)+2
V(g)$color <- kol
tkplot(g)
```

Wynik działania algorytmu można obejrzeć na rysunku 5.4. Obiekt `t` jest drzewem o 15 wierzchołkach, w którym każdy wierzchołek ma co najwyżej 3 dzieci. Oczywiście algorytm zadziała błędnie, jeśli dany graf nie będzie



Rysunek 5.4: Graf dwudzielny pokolorowany optymalnie

grafem dwudzielnym.

Rozdział 6

Problem komiwojażera

Definicja 1: Definicja problemu komiwojażera

Dla grafu $G = (V, E)$ pełnego, ważonego i nieskierowanego należy znaleźć cykl Hamiltona h, którego łączna waga $w(h)$ jest najmniejsza.

Definicja 2: Definicja asymetrycznego problemu komiwojażera

Dla grafu $G = (V, E)$ pełnego, ważonego i skierowanego należy znaleźć cykl Hamiltona h, którego łączna waga $w(h)$ jest najmniejsza.

Algorytm 4: Algorytm generujący kolejne permutacje

Wejście: permutacja π^i.
Wyjście: kolejna permutacja w stosunku do π^i.

Krok 1 Ustaw $\pi := \pi^i$.

Krok 2 Znajdź największe j takie, że $1 \leqslant j \leqslant n-1$ i $\pi_j < \pi_{j+1}$
Jeżeli znaleziono j, to:

- Znajdź największe k takie, że $j < k \leqslant n$ i $\pi_j < \pi_k$;
- Zamień miejscami π_j oraz π_k.

- Odwróć porządek elementów $\pi_{j+1}, \ldots, \pi_n$.

6.1 Metoda przeglądania wszystkich możliwych permutacji miast

Algorytm generujący następną permutację wraz z przykładem wywołania.

```
perm <- function(x){
  y <- c(x[-1],0)
  j <- max(which(x<y), 0)

  if (j == 0){
    cat("To juz ostatnia permutacja :) \n")
    return(x)
  }else{
    k <- max(which(x>x[j]))
    x[c(j,k)] <- x[c(k,j)]
    y <- c(x[1:j], rev(x[(j+1):length(x)]))
    return(y)
  }
}

x<- c(1,2,3,4,5)
perm(x)

# [1] 1 2 3 5 4
```

Wykorzystamy algorytm generowania kolejnej permutacji do przejrzenia wszystkich możliwych tras komiwojażera i wybraniu najkrótszej. Za pomocą macierzy sąsiedztwa A dany mamy graf pełny, ważony.

```
> A
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    3    4    2    7
[2,]    3    0    4    6    3
[3,]    4    4    0    5    8
[4,]    2    6    5    0    6
```

```
[5,]    7    3    8    6    0
```

Ustawiamy hipotetyczną, gorszą od najgorszej możliwej długość cyklu komiwojażera.

```
dlugosc <- 1000
```

Ustawiamy początkowy cykl i początkową permutację x. Wszystkich tras do przejrzenia jest $5! = 120$.

```
x <- c(1,2,3,4,5)
cykl <- c(1,2,3,4,5)
for(i in 1:120){
  dl <- 0
  for(k in 1:4)
  {
    dl = dl + A[x[k],x[k+1]]
  }
  dl = dl + A[x[5],x[1]]
  if(dlugosc > dl){
    dlugosc <- dl
    cykl <-x
    cat(x," dlugosc: ",dlugosc, "\n")
  }
  x<- perm(x)
}
```

Każdą lepszą trasę komiwojażera wypisujemy na ekranie, efekt działania jest następujący.

```
1 2 3 4 5  dlugosc:  25
1 2 3 5 4  dlugosc:  23
1 2 5 3 4  dlugosc:  21
1 3 2 5 4  dlugosc:  19
To juz ostatnia permutacja :)
```

Zatem optymalna trasa komiwojażera ma długość 19.

6.2 Komiwojażer i losowe wstawianie

Przeglądanie całego zbioru możliwych tras komiwojażera jest działaniem o superwykładniczej złożoności obliczeniowej. Z tego powodu konstruuje się algorytmy przybliżone o złożoności wielomianowej. Nie dają one pewności otrzymania optymalnego rozwiązania, ale pewne jego przybliżenie.

Metoda losowego wstawiania do cyklu jest jedną z prostszych do zaprogramowania i w praktyce okazuje się dawać dobre rezultaty.

Pokażemy na przykładzie grafu miast, których współrzędne znajdują się w pliku `dj38.tsp.txt` jego działanie. Pierwszych 10 linijek pliku:

```
1 11003.611100 42102.500000
2 11108.611100 42373.888900
3 11133.333300 42885.833300
4 11155.833300 42712.500000
5 11183.333300 42933.333300
6 11297.500000 42853.333300
7 11310.277800 42929.444400
8 11416.666700 42983.333300
9 11423.888900 43000.277800
10 11438.333300 42057.222200
```

Pierwsza kolumna to numer kolejny miasta, a kolejne dwie to współrzędne miasta. Dane z pliku wczytujemy do R, a następnie przekształcamy w macierz sąsiedztwa.

```
x <- read.table("dj38.tsp.txt")
x <- structure(as.matrix(x), dimnames=NULL)

zrobMacierzOdleglosci <- function(x){
  A <- matrix(0, nrow(x), nrow(x))
  for(i in 1:nrow(x)){
    for(j in 1:nrow(x)){
      A[i,j] <- sqrt((x[i,2]-x[j,2])^2 + (x[i,3]-x[j,3])^2)
    }
  }
```

```
  A
}
```

Funkcja `wstawianieDoCyklu` wstawia losowo wybrany wierzchołek w najbardziej odpowiednie w danym momencie miejsce. Jest ono zdefiniowane jako miejsce, w którego wstawienie nowego wierzchołka spowoduje najmniejszy przyrost długości cyklu. Cała funkcja wykonuje się dla dwudziestu różnych losowań kolejności wierzchołków. Na koniec funkcja porównuje otrzymane przybliżone rozwiązanie z rozwiązaniem optymalnym.

```
wstawianieDoCyklu <- function(A){
  for(k in 1:20){
    wektorLos <- sample(1:nrow(A))
    cykl <- c(wektorLos[1:3], wektorLos[1])

    for (j in 4:nrow(A)){
      minimum <- 100000000
      gdzie <- 0

      for(i in 1:(j-1)){
        u <- A[cykl[i],wektorLos[j]]+
          A[cykl[i+1],wektorLos[j]]-
          A[cykl[i],cykl[i+1]]
        if(u < minimum){
          minimum <- u
          gdzie <- i
        }
      }
      cykl <- append(cykl, wektorLos[j], after = gdzie)
    }
    dlugosc <- 0
    for (i in 1:nrow(A)){
      dlugosc <- dlugosc + A[cykl[i],cykl[i+1]]
    }
    cat ("\n\nCykl: ", cykl, "\ndlugosc: ", dlugosc,
         "\n gorszy o", ((dlugosc-6656)/6656)*100,"%")
```

```
  }
}
```

Przykładowy wynik działania programu.

```
Cykl:  5 7 13 15 8 9 11 12 16 17 18 19 27 31 36 34 33 38 37
      35 28 24 22 20 23 25 26 30 32 29 21 14 10 1 2 4 6 3 5
dlugosc:  7196.669
 gorszy o 8.123032 %

Cykl:  4 3 5 6 7 13 15 8 9 12 11 16 17 18 19 22 24 28 27 31
      36 34 33 38 37 35 32 30 29 26 25 23 20 21 14 10 1 2 4
dlugosc:  7445.931
 gorszy o 11.86795 %

Cykl:  6 5 3 4 2 1 10 14 21 29 30 32 35 37 38 33 34 36 31 27
      28 24 22 25 26 23 20 15 13 19 18 17 16 12 11 9 8 7 6
dlugosc:  6664.114
 gorszy o 0.1218986 %
```

6.3 Podwajanie krawędzi

```
podwajanieKrawedzi <- function(A){
  gg <- graph_from_adjacency_matrix(A,
        weighted = TRUE, "undirected")
  mgg <- mst(gg)

  for(k in 1: gorder(mgg)){
    mmgg <- 2* as_adjacency_matrix(mgg, sparse = FALSE)
    lista <- c()
    bierz <- k
    nbierz <- bierz

    for (j in 1:(2*(ncol(mmgg)-1))){
      lista <- c(lista, bierz)
      for (i in 1: ncol(mmgg)){
```

```
        if (mmgg[bierz,i]==2){
          mmgg[bierz,i] <- 1
          mmgg[i,bierz] <- 1
          nbierz <- i
          break
        }
      }
      if (nbierz == bierz){
        for (i in 1: ncol(mmgg)){
          if (mmgg[bierz,i]==1){
            mmgg[bierz,i] <- 0
            mmgg[i,bierz] <- 0
            nbierz <- i
            break
          }
        }
      }
      bierz <- nbierz
    }

    komiw <- unique(lista)
    komiw <- c(komiw, komiw[1])
    waga <- 0
    for(i in 1:(length(komiw)-1)){
      waga <- waga + A[komiw[i],komiw[i+1]]
    }
    cat ("Cykl Hamiltona:", komiw)
    cat(" Jego waga wynosi:", waga)
    cat("\n")
  }
}
```

Rozdział 7

Różne

7.1 Drzewo losowe

Poniżej przestawiamy funkcję generującą losowe drzewo ukorzenione w liściu.

Na początku algorytm tworzy puste drzewo. Następnie dodaje do niego podaną jako argument funkcji liczbę wierzchołków. Dwie krawędzie, a mianowicie $(1, 2)$ oraz $(2, 3)$ dodawane są „na sztywno" aby mieć pewność, że wierzchołek 1, czyli nasz korzeń, będzie miał stopień 1. W pętli losowane są kolejne krawędzie. W powstałym w ten sposób drzewie dziecko wierzchołka v ma zawsze większy numer v.

```
generuj <- function(ile){
  T <- make_tree(0)
  T <- add_vertices(T, ile, color = 2, sta = 0, code = 0)
  T <- add.edges(T, c(1,2))
  T <- add.edges(T, c(2,3))
  for (i in c(4:ile)){
    T <- add.edges(T, c( sample(2:(i-1),1) ,i))
  }
  return(T)
}
```

Rozdział 8

Inne

Spis rysunków