

Тема 13

**Графи. Програми на Haskell за намиране
на пътища в граф**

Основни дефиниции

Граф Γ се нарича наредената двойка $\langle V, R \rangle$, където $V = \{a_1, a_2, \dots, a_n\}$ е множество, R е ненареден списък от двойки елементи на V .

Елементите на V се наричат **върхове** (**възли**) на графа Γ , а елементите на R – **ребра** (**дъги**) на графа Γ .

Ако ребрата на Γ са наредени двойки, графът Γ се нарича *ориентиран*; ако ребрата не са наредени, Γ се нарича *неориентиран* граф.

Геометричното изображение на графите се получава, като:

- изберем толкова различни точки, колкото върха има графът, и съпоставяме на всеки връх по една точка;
- върховете a_i и a_j съединим с:
 - линия, ако графът е неориентиран и $(a_i, a_j) \in R$;
 - стрелка, ако графът е ориентиран и $(a_i, a_j) \in R$.

Основно понятие в теорията на графите е понятието **път**.

Под *път в ориентиран граф* се разбира всяка редица от ребра, имаща вида

$$\langle a_{i_1}, a_{i_2} \rangle, \langle a_{i_2}, a_{i_3} \rangle, \dots, \langle a_{i_{l-1}}, a_{i_l} \rangle, \langle a_{i_l}, a_{i_{l+1}} \rangle.$$

Ще казваме, че този път води от a_{i_1} до $a_{i_{l+1}}$ (има начало a_{i_1} и край $a_{i_{l+1}}$) и има *дължина* l .

Под *път в неориентиран граф* се разбира всяка редица от ребра, имаща вида

$$\langle a_1, a_2 \rangle, \langle b_1, b_2 \rangle, \dots, \langle e_1, e_2 \rangle, \langle l_1, l_2 \rangle,$$

където единият елемент на всяко ребро принадлежи на левия съсед, а другият елемент – на десния съсед. От крайните ребра се изисква само по един техен елемент да принадлежи на съседа на реброто. Другите два елемента определят *краищата* на пътя, а броят на ребрата – неговата *дължина*.

Забележка. Често пътищата в графа се представят като поредици от участващите в тях (съставлящите ги) върхове (възли).

Например пътят

$\langle a_{i_1}, a_{i_2} \rangle, \langle a_{i_2}, a_{i_3} \rangle, \dots, \langle a_{i_{l-1}}, a_{i_l} \rangle, \langle a_{i_l}, a_{i_{l+1}} \rangle$ в ориентирувания граф Γ се представя чрез поредицата (списъка) от върхове $a_{i_1}, a_{i_2}, \dots, a_{i_l}, a_{i_{l+1}}$.

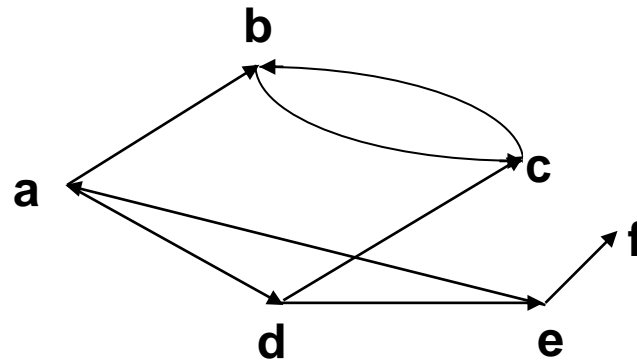
Път със съвпадащи краища (съвпадащи начало и край) се нарича *цикъл*. Път, който съдържа цикъл, се нарича *цикличен*; в противен случай пътят се нарича *ацикличен*.

Графът $\Gamma = \langle V, R \rangle$ се нарича *краен*, ако V и R са крайни.

Представяне на (ориентирани) графи със средствата на езика Haskell

Най-често графът се представя чрез списък, елементите на който съответстват на ребрата (дъгите), които започват от съответните върхове (възли) на този граф.

Примерен граф:



Фиг. 1

Примерни представяния на графа от фиг. 1:

```
[ ("a", ["b", "d"]), ("b", ["c"]), ("c", ["b"]),  
  ("d", ["c", "e"]), ("e", ["a", "f"])]
```

```
[ ("a", ["b", "d"]), ("b", ["c"]), ("c", ["b"]),  
  ("d", ["c", "e"]), ("e", ["a", "f"]), ("f", [])]
```

Второто представяне е по-удобно в случаите, когато е необходимо да се работи със списъка от върховете на графа или поне с техния брой. То е по-целесъобразно и в случаите, когато в графа има „изолирани“ върхове (такива върхове, от които не започват и в които не завършват дъги от графа).

**Дефиниции на някои функции
за работа с (ориентирани) графи.
Намиране на път в граф**

```
type Node = String
```

```
type Graph = [(Node, [Node])]
```

```
type Path = [Node]
```

```
graph1 :: Graph
```

```
graph1 = [ ("a", ["b", "d"]), ("b", ["c"]), ("c", ["b"]),  
           ("d", ["c", "e"]), ("e", ["a", "f"]) ]
```



```
graph2 :: Graph
graph2 = [ ("a", ["b", "d"]), ("b", ["c"]), ("c", ["b"]),
           ("d", ["c", "e"]), ("e", ["a", "f"]), ("f", []) ]

-- graph1 и graph2 са различни представяния на
-- примерния граф от фиг. 1. graph1 включва само
-- елементи, които имат за ключове такива върхове,
-- от които започват ребра в графа; graph2 има
-- за ключове на елементите си всички върхове
-- на графа (вкл. в „изолираните“ върхове, ако има
-- такива) .
```

```

assoc :: Eq a => a -> [(a,[b])] -> (a,[b])
-- assoc :: Node -> [(Node,[Node])] -> (Node,[Node])
-- Асоциативно търсене в списък от двойки по даден ключ.
assoc key [] = (key,[])
assoc key (x:xs)
    | fst x==key = x
    | otherwise  = assoc key xs

successors :: Node -> Graph -> [Node]
successors node graph = snd (assoc node graph)

```

```

is_a_node :: Node -> Graph -> Bool
-- Проверява дали node е връх в графа graph.
-- Работи и при двете коментирани представяния на графа.
is_a_node node graph = rt node || lf node
  where rt :: Node -> Bool
        rt x = elem x (map fst graph)
        lf :: Node -> Bool
        lf x = elem x (concat (map snd graph))

is_a_path :: Path -> Graph -> Bool
-- Проверява дали даден списък от „върхове“ е път
-- в даден граф.
is_a_path [] _ = True
is_a_path [node] graph = is_a_node node graph
is_a_path (n1:(n2:others)) graph
  | elem n2 (successors n1 graph)
    = is_a_path (n2:others) graph
  | otherwise = False

```

```
correct_path :: Path -> Graph -> Bool
-- Проверява дали даден списък от „върхове“ е
-- ацикличен път в даден граф.
correct_path path graph = (is_a_path path graph)
                           && (not (cycled path))

cycled :: Path -> Bool
-- Проверява дали в даден „път“ се съдържа цикъл.
cycled [] = False
cycled (n:ns)
  | elem n ns = True
  | otherwise = cycled ns
```

```
gen_next :: Path -> Graph -> [Path]
-- Връща като резултат списък от продълженията
-- на даден път.
gen_next path graph = map (\x -> (x:path))
    (successors (head path) graph)

generate_paths :: [Path] -> Graph -> [Path]
-- Връща като резултат списък от продълженията
-- на пътищата от даден списък.
generate_paths paths graph
    = concat (map (\x -> gen_next x graph) paths)
```

```
connected :: Node -> Node -> Int -> Graph -> Bool
-- Проверява дали съществува път в даден граф graph
-- от върха node1 до върха node2, който има дължина,
-- не по-голяма от n.
connected node1 node2 n graph
  | node1==node2 = True
  | n<=0         = False
  | otherwise    = connect1 [[node1]] node2 n graph
```

```
connect1 :: [Path] -> Node -> Int -> Graph -> Bool
connect1 paths node n graph
  | n<=0          = False
  | null paths    = False
  | otherwise     = if (elem node (map head lnp)) then True
                     else connect1 lnp node (n-1) graph
    where lnp :: [Path]
          lnp = generate_paths paths graph
```

```
assoc1 :: Eq a => a -> [[a]] -> [a]
-- Асоциативно търсене в списък от списъци
-- по даден ключ.
assoc1 _ [] = []
assoc1 key (x:xs)
  | head x==key = x
  | otherwise   = assoc1 key xs
```



```

path :: Node -> Node -> Graph -> Path
-- Намира път в даден граф graph от върха node1
-- до върха node2. Работи с графи, в които явно
-- са посочени „синовете“ на всички възли.
-- Основава се на идеята, че ако съществува път
-- между два върха, то съществува и ацикличен път
-- между тези върхове, а дължината на най-дългия
-- ацикличен път в графа не може да надхвърли n-1,
-- където n е броят на върховете в графа.
path node1 node2 graph
  | node1==node2 = [node1]
  | otherwise    = build_path [[node1]] node2
                        (length graph-1) graph

```

```

build_path :: [Path] -> Node -> Int -> Graph -> Path
build_path paths node n graph
  | n<=0          = []
  | null paths    = []
  | otherwise     = if (null pth)
    then build_path lnp node (n-1) graph
    else reverse pth
    where lnp :: [Path]
          lnp = generate_paths paths graph
          pth :: Path
          pth = assoc1 node lnp

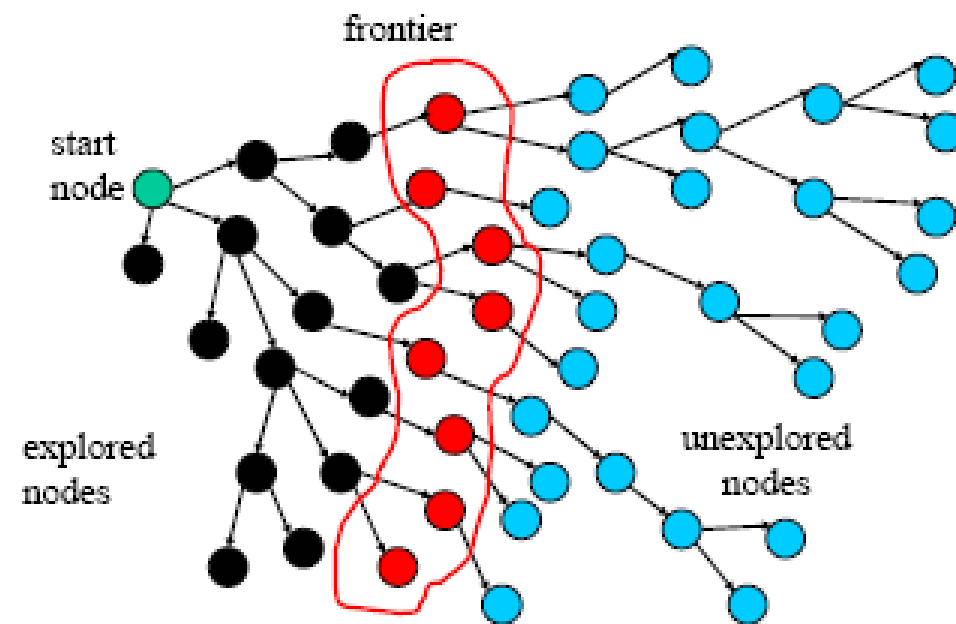
```

Забележка. Дефинираните по-горе функции за намиране на път в граф реализират един „наивен“ метод, който е добра илюстрация на използването на функции от по-висок ред при работа със съставни структури от данни (при моделирането на графи чрез списъци).

Основни стратегии на търсене на път в граф

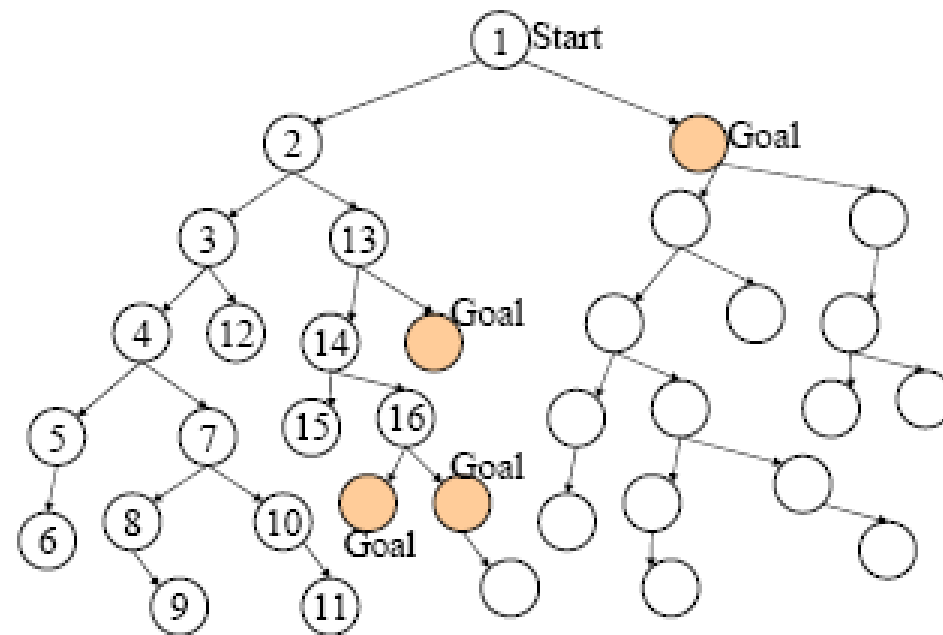
Общ алгоритъм за търсене:

- Даден е граф; известни са началният възел (Start) и целевият възел (Goal);
- Последователно се изследват пътищата от началния възел;
- Поддържа се фронт/граница (frontier) от пътищата, които са били изследвани;
- По време на процеса на търсене фронтът се разширява в посока към неизследваните възли, докато се достигне до целевия възел;
- Начинът, по който фронтът се разширява, както и това, точно кой път от фронта се избира за разширяване на фронта на следващата стъпка, дефинира *стратегията на търсене (search strategy)*.



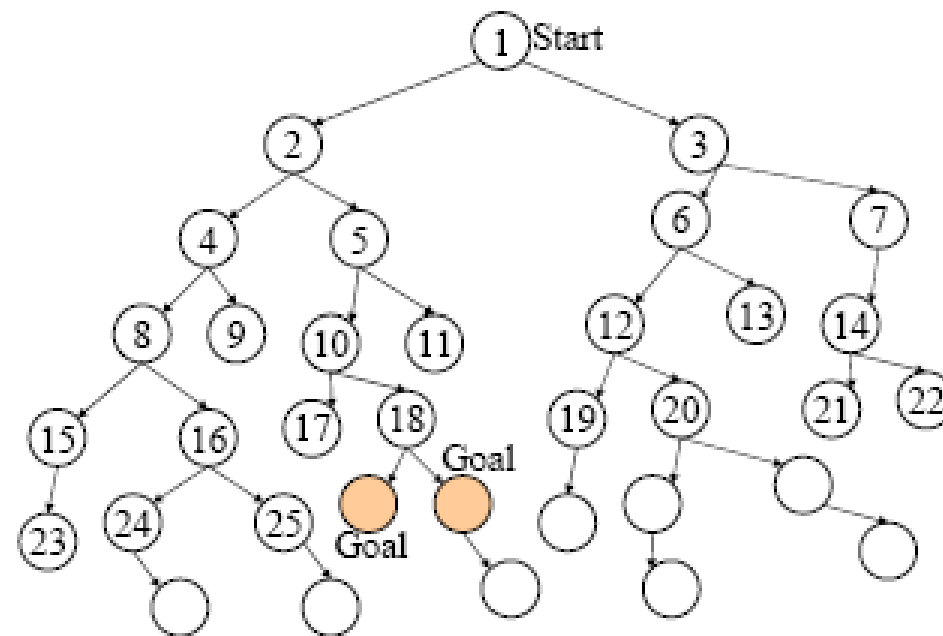
Търсене в дълбочина (depth-first search)

- При търсенето в дълбочина фронтът се обработва като стек.
- В началото на търсенето фронтът има вида **[[Start]]**.
- Ако на текущата стъпка фронтът е празен (**[]**), търсенето се прекратява и се връща **[]** (индикация за липса на решение).
- Ако на текущата стъпка фронтът е **[p1, p2, ... , pn]**, то:
 - избира се първият елемент на фронта (**p1**);
 - ако пътят **p1** е достигнал целта, търсенето се прекратява и **p1** се връща като резултат;
 - ацикличните пътища **p1'**, **p1''**, ... , **p1(k)**, които продължават с една дъга **p1**, се добавят в началото на стека (преди **p2**), т.е. фронтът придобива вида **[p1', p1'', ... , p1(k), p2, ... , pn]** и се преминава към следващата стъпка;
 - **p2** се обработва едва след като се изследват всички пътища, които са продължения на **p1**.



Търсене в широчина (breadth-first search)

- При търсенето в широчина фронтът се обработва като опашка.
- В началото на търсенето фронтът има вида **[[Start]]**.
- Ако на текущата стъпка фронтът е празен (**[]**), търсенето се прекратява и се връща **[]** (индикация за липса на решение).
- Ако на текущата стъпка фронтът е **[p1, p2, ... , pn]**, то:
 - избира се първият елемент на фронта (**p1**);
 - ако пътят **p1** е достигнал целта, търсенето се прекратява и **p1** се връща като резултат;
 - ацикличните пътища **p1'**, **p1''**, ... , **p1(k)**, които продължават с една дъга **p1**, се добавят в края на опашката (след **pn**), т.е. фронтът придобива вида **[p2, ... , pn, p1', p1'', ... , p1(k)]** и се преминава към следващата стъпка;
 - **p1'** се обработва едва след като се изследват всички пътища **p2, ... , pn**.
- Намира най-краткия път от началния до целевия възел.



Програмна реализация

```
type Node = String
```

```
type Graph = [(Node, [Node])]
```

```
type Path = [Node]
```

```
graph :: Graph
```

```
-- Примерен граф.
```

```
graph = [ ("a", ["b", "c", "d"]), ("b", ["e", "f"]),  
          ("c", ["g", "i"]), ("d", ["f", "h"]),  
          ("e", ["i"]), ("f", ["j"]), ("h", ["j"]) ]
```

```

assoc :: Eq a => a -> [(a,[b])] -> (a,[b])
-- assoc :: Node -> [(Node,[Node])] -> (Node,[Node])
-- Асоциативно търсене в списък от двойки
-- по даден ключ (същото както по-горе) .
assoc key [] = (key,[])
assoc key (x:xs)
    | fst x==key = x
    | otherwise  = assoc key xs

successors :: Node -> Graph -> [Node]
successors node graph = snd (assoc node graph)

```

```
extend :: Path -> Graph -> [Path]
-- Разширяване на фронта.
extend path graph = concat
  (map (\x -> if (elem x path) then [] else [x:path])
    (successors (head path) graph))
```

```

depth_first_search :: Node -> Node -> Graph -> Path
-- Търсене в дълбочина.
depth_first_search node1 node2 graph = reverse
    (depth_first [[node1]] node2 graph)

depth_first :: [Path] -> Node -> Graph -> Path
depth_first [] _ _ = []
depth_first (path:others) goal graph
    | goal==head path = path
    | otherwise       = depth_first
        ((extend path graph)++others) goal graph

```

```

breadth_first_search :: Node -> Node -> Graph -> Path
-- Търсене в широчина.
breadth_first_search node1 node2 graph = reverse
    (breadth_first [[node1]] node2 graph)

breadth_first :: [Path] -> Node -> Graph -> Path
breadth_first [] _ _ = []
breadth_first (path:others) goal graph
    | goal==head path = path
    | otherwise       = breadth_first
        (others++(extend path graph)) goal graph

```

Примери

`depth_first_search "a" "i" graph →`
`["a", "b", "e", "i"]`

`breadth_first_search "a" "i" graph →`
`["a", "c", "i"]`