

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Иркутский государственный университет»
(ФГБОУ ВО «ИГУ»)
Институт математики и информационных технологий
Кафедра информационных технологий

ОТЧЕТ
по курсовой работе

**РЕАЛИЗАЦИЯ АЛГОРИТМА ФЛОЙДА-УОРШАЛЛА НА ЯЗЫКЕ
ПРОГРАММИРОВАНИЯ HASKELL**

Студента 3 курса группы 02321-ДБ
направления 01.03.02 «Прикладная
математика и информатика»
Саблина Михаила Александровича

Руководитель:
К. т. н., доцент
Черкашин Евгений Александрович
Оценка _____

Иркутск – 2022

СОДЕРЖАНИЕ

1	Введение	3
2	Теоретические основы	4
3	Реализация	5
4	Тестирование алгоритма	7
	ЗАКЛЮЧЕНИЕ	8
	СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	9
	ПРИЛОЖЕНИЕ А Исходный код	10

1 Введение

Haskell – функциональный язык программирования, сильно отличающийся от императивных и смешанных языков разработки. Важной особенностью языка является поддержка ленивых вычислений, что позволяет ускорить работу программы. В Haskell аргументы функции вычисляются только тогда, когда действительно требуются для вычисления. Причем ленивые вычисления выполняются без вмешательства самого программиста.

Haskell – актуален. Язык применяется в финансовом секторе – для разработки собственных инструментов в крупных банках и других компаниях. Также Haskell часто используется для обработки текстов, синтаксического анализа, а также веб-разработки.

Целью данной курсовой работы является ознакомление с Haskell’ем посредством решения задач, связанных с графами, укрепление и развитие навыков программирования.

2 Теоретические основы

Алгоритм Флойда-Уоршалла:

Шаг 0. Определяем начальную матрицу расстояний D_0 и матрицу последовательности узлов S_0 . Диагональные элементы обеих матриц помечаются знаком «-», показывающим, что эти элементы в вычислениях не участвуют. Полагаем $k=1$.
Основной шаг k . Задаем строку k и столбец k как ведущую строку и ведущий столбец. Рассматриваем возможность применения треугольного оператора ко всем элементам d_{ij} матрицы D_{k-1} . Если выполняется равенство

$$d_{ik} + d_{kj} < d_{ij}, (i \neq k, j \neq k, i \neq j)$$

Тогда выполняем следующие действия:

- А) создаем матрицу D_k путем замены в матрице D_{k-1} элемента d_{ij} на сумму $d_{ik} + d_{kj}$,
- Б) создаем матрицу S_k путем замены в матрице S_{k-1} элемента s_{ij} на k . Полагаем $k=k+1$ и повторяем шаг k .

После реализации n -шагов алгоритмов:

1. Расстояние между узлами i и j равно элементу d_{ij} в матрице D_n .
2. Промежуточные узлы пути от узла i к узлу j определяем по матрице S_n . Пусть $s_{ij} = k$, тогда имеем путь $i \rightarrow k \rightarrow j$. Если далее $s_{ik} = k$ и $s_{kj} = j$, тогда считаем, что весь путь определен, так как найдены все промежуточные узлы. В противном случае повторяем описанную процедуру для путей от узла i к узлу k и от узла k к узлу j .
Алгоритм Флойда-Уоршалла отлично подходит для задач, в которых нужно найти путь между двумя любыми узлами.

3 Реализация

Сначала мы определяем общий тип данных для представления кратчайшего пути. Тип `a` представляет расстояние. Это может быть число в случае взвешенного графа или логическое значение только для ориентированного графа. Тип `b` подходит для меток вершин (целые числа, символы, строки...)

```
data Shortest b a = Shortest { distance :: a, path :: [b] }  
    deriving Show
```

Далее заметим, что кратчайшие пути образуют полугруппу со следующим правилом «сложения»:

```
instance (Ord a, Eq b) => Semigroup (Shortest b a) where  
    a < b = case distance a `compare` distance b of  
        GT -> b  
        LT -> a  
        EQ -> a { path = path a `union` path b }
```

Он находит минимальный путь по расстоянию, а при равенстве расстояний соединяет оба пути. Поднимем эту полугруппу до моноида с помощью обертки `Maybe`.

Граф представлен в виде карты, содержащей пары вершин и соответствующие им веса. Таблица расстояний представляет собой карту, содержащую пары совместных вершин и соответствующих кратчайших путей.

Теперь мы готовы определить основную часть алгоритма Флойда-Уоршалла, который обрабатывает должным образом подготовленную таблицу расстояний `dist` для заданного списка вершин `v`:

```
floydWarshall v dist = foldr innerCycle (Just <$> dist) v  
    where
```

```

innerCycle k dist = (newDist <$> v <*> v) `setTo` dist
where
  newDist i j =
    ((i,j), do a <- join $ lookup (i, k) dist
              b <- join $ lookup (k, j) dist
              return $ Shortest (distance a <◇> distance b) (path a))

  setTo = unionWith (<◇>) . fromList

```

floydWarshall производит только первые шаги кратчайших путей. Целые пути строятся с помощью следующей функции:

```

buildPaths d = mapWithKey (\pair s → s { path = buildPath pair}) d
where
  buildPath (i,j)
    | i == j    = [[j]]
    | otherwise = do k <- path $ fromJust $ lookup (i,j) d
                    p <- buildPath (k,j)
                    [i : p]

```

Всю пре- и постобработку выполняет основная функция findMinDistances:

```

findMinDistances v g =
  let weights = mapWithKey (\(_,j) w → Shortest w [j]) g
      trivial = fromList [ ((i,i), Shortest mempty []) | i <- v ]
      clean d = fromJust <$> filter isJust (d \\\ trivial)
  in buildPaths $ clean $ floydWarshall v (weights <◇> trivial)

```

Вспомогательная функция:

```

showShortestPaths v g = mapM_ print $ toList $ findMinDistances v g

```

4 Тестирование алгоритма

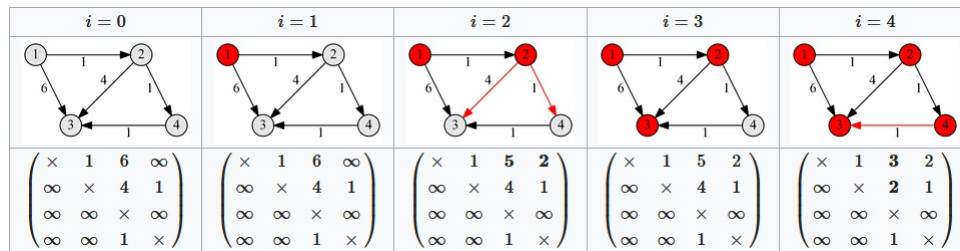


Рисунок 4.1 – В качестве примера возьмем данный граф

В переменной `g` мы передаем все возможные пути, функцией `showSortestPath` выводим в консоль результат

```
main = do
  let g = fromList [((1,2), 1)
                    ,((1,3), 6)
                    ,((2,3), 4)
                    ,((2,4), 1)
                    ,((4,3), 1)]

  showShortestPaths [1..4] (Sum <$> g)
```

Результат в консоли

```
ghci> main
((1,2),Shortest {distance = Sum {getSum = 1}, path = [[1,2]]})
((1,3),Shortest {distance = Sum {getSum = 3}, path = [[1,2,4,3]]})
((1,4),Shortest {distance = Sum {getSum = 2}, path = [[1,2,4]]})
((2,3),Shortest {distance = Sum {getSum = 2}, path = [[2,4,3]]})
((2,4),Shortest {distance = Sum {getSum = 1}, path = [[2,4]]})
((4,3),Shortest {distance = Sum {getSum = 1}, path = [[4,3]]})
```

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы был рассмотрен функционал языка программирования Haskell для решения задачи нахождения кратчайшего пути на графе с помощью алгоритма Флойда-Уоршалла.

Показана реализация данного алгоритма и его корректная работа. При выполнении курсовой были закреплены и показаны навыки работы с данным языком программирования.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. 1. Х.А. Таха Введение в исследование операций. М.: Издательский дом "Вильямс 2005. 259-260 с.
2. 2. М. Липовач Изучай Haskell во имя добра 2012

ПРИЛОЖЕНИЕ А Исходный код

```
import Control.Monad (join)
import Data.List (union)
import Data.Map hiding (foldr, union)
import Data.Maybe (fromJust, isJust)
import Data.Semigroup
import Prelude hiding (lookup, filter)

data Shortest b a = Shortest {distance :: a, path :: [b]} deriving Show

instance (Ord a, Eq b) => Semigroup (Shortest b a) where
  a <math>\diamond</math> b = case distance a `compare` distance b of
    GT -> b
    LT -> a
    EQ -> a { path = path a `union` path b }

floydWarshall v dist = foldr innerCycle (Just <math>\diamond</math> dist) v
  where
    innerCycle k dist = (newDist <math>\diamond</math> v <math>\ltimes</math> v) `setTo` dist
      where
        newDist i j =
          ((i,j), do a <- join $ lookup (i, k) dist
                    b <- join $ lookup (k, j) dist
                    return $ Shortest (distance a <math>\diamond</math> distance b) (path a))

        setTo = unionWith (<math>\diamond</math>) . fromList

buildPaths d = mapWithKey (\pair s -> s { path = buildPath pair}) d
  where
    buildPath (i,j)
      | i == j      = [[j]]
      | otherwise = do k <- path $ fromJust $ lookup (i,j) d
                      p <- buildPath (k,j)
```

```
[i : p]
```

```
findMinDistances v g =
```

```
  let weights = mapWithKey (\(_,j) w → Shortest w [j]) g
      trivial = fromList [ ((i,i), Shortest mempty []) | i ← v ]
      clean d = fromJust <$> filter isJust (d \\\ trivial)
  in buildPaths $ clean $ floydWarshall v (weights <◇> trivial)
```

```
showShortestPaths v g = mapM_ print $ toList $ findMinDistances v g
```

```
main = do
```

```
  let g = fromList [((1,2), 1)
                    ,((1,3), 6)
                    ,((2,3), 4)
                    ,((2,4), 1)
                    ,((4,3), 1)]
```

```
  showShortestPaths [1..4] (Sum <$> g)
```