Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет информационных технологий и управления

ОТЧЕТ

по лабораторной работе № 1

на тему: «Проектирование компонента решателя задач на основе декларативных моделей решения задач»

Выполнил:

Студент группы 426401

Махмудов Асадулло Кобилжон угли

Проверил:

Минск 2025

**Выбор модели (функциональный подход)**

Было принято решение использовать функциональную модель для решения задачи поиска кратчайшего пути в графе, потому что этот подход наглядно демонстрирует ключевые свойства декларативного программирования: чистые функции, отсутствие побочных эффектов и неизменяемость данных. В функциональном стиле весь алгоритм представляется как композиция небольших, простых для понимания и тестирования, функций. При этом реализация алгоритма Дейкстры в Haskell (или аналогичном языке) получается лаконичной: хранение расстояний, множеств необработанных вершин и предшественников оформляется через неизменяемые структуры («Map» и «Set»), а рекурсивный обход заменяет привычный императивный цикл.

Сравнивая с другими декларативными парадигмами (логической и продукционной), функциональная модель для графовых алгоритмов оказывается наиболее естественной: мы имеем дело с математи­ческой моделью (графом) и с вычислением функций на этой модели (расчёт расстояний и пути). В Prolog‑реализации потребовались бы дополнительные ухищрения для эффективного поиска оптимума, а в продукционном движке (например, CLIPS) – имитация приоритетной очереди через правила. Функциональный же код сразу отражает «желание» получить минимумы и обновлять их рекурсивно.

Кроме того, реализация на Haskell позволяет минимизировать объём «бокового» кода: IO‑часть (чтение входных данных и вывод результата) отделяется от ядра алгоритма, который не имеет побочных эффектов и легко тестируется «из командной строки» или в интерактивном режиме GHCi. Это упрощает сопровождение и демонстрацию работы алгоритма при сдаче лабораторной.

**Постановка задачи**

Необходимо разработать программу (компонент «решателя») для нахождения кратчайшего пути между двумя заданными вершинами в неориентированном взвешенном графе.

Входные данные:

1. Список рёбер графа, где каждое ребро задаётся тройкой «(u, v, w)», означающей соединение вершины «u» с вершиной «v» с весом «w» (целое положительное число).

2. Пара вершин «(start, goal)», между которыми нужно найти кратчайший маршрут.

Выходные данные:

* Либо сообщение о том, что путь не существует (если вершины «star» и «goal» оказываются в разных компонентах),
* Либо длина кратчайшего пути (сумма весов рёбер) и сам список вершин [start = v₀, v₁, …, vₖ = goal], по которому этот минимум достигается.

Граф считается статическим, без петель (ребро из вершины в себя) и без кратных одинаковых рёбер. Все вершины нумеруются целыми числами от 1 до N..

**Алгоритм решения**

1. Инициализация структур:

* Построить отображение («Map») «вершина → список соседей с весами», используя данные о рёбрах.
* Задать для каждой вершины значение расстояния «dist(v)»:
* Для «start« - 0.
* Для всех остальных - бесконечность (или очень большое число).
* Инициализировать отображение предшественников «prev(v)», пока пустое.
* Сформировать множество «unvisited», содержащее все вершины графа.

2. Основной цикл (пока есть непосещённые вершины):

1. Выбрать из «unvisited» вершину «u» с минимальным текущим расстоянием «dist(u)».

* Если «dist(u) = ∞» или «u = goal», прервать цикл.

2. Для каждого соседа «(v, w)« вершины «u» выполнить «релаксацию»:

* Вычислить альтернативное расстояние «alt = dist(u) + w».
* Если «alt < dist(v)», то обновить «dist(v) = alt2» и запомнить «prev(v) = u».

3. Удалить «u» из множества «unvisited».

3. Проверка результата:

* Если после цикла «dist(goal) = ∞», путь отсутствует.
* Иначе нужно восстановить сам маршрут, начиная с «goal» и идя по цепочке предшественников:

path = []

current = goal

while current ≠ start:

path = current : path

current = prev(current)

path = start : path

Тогда «path» будет содержать список вершин от «start» до «goal» по кратчайшему пути, а «dist(goal)» даст суммарный вес.

4. Сложность:

* При простом хранении «unvisited» в виде множества и линейном поиске минимума: ~O(V² + E), где V – число вершин, E – число рёбер.
* При использовании мин‑кучи (например, «Data.PSQueue» или «Data.Heap») можно снизить до O((V + E) log V). В нашей учебной реализации достаточно простого поиска минимума, так как предполагается небольшое количество вершин (до сотен).

Таким образом, суть решения:

* Декларативно (в функциональном стиле) задать неизменяемые структуры данных «Map» и «Set», описать чистую функцию, которая в каждую итерацию выбирает «текущую» вершину и рекурсивно обновляет отображения «dist» и «prev».
* В конце вернуть либо «Nothing» (нет пути), либо «Just (path, dist)», где «path» – список вершин, «dist» – длина.

**Решение**

1. Среда разработки и организация проекта

Для компиляции и запуска использован компилятор GHC версии 9.8.2, установленный в папке C:\tools\ghc-9.8.2\bin (поле PATH в системных переменных содержит этот путь).

Проект собран без использования Stack/Cabal - напрямую через команду GHC.

Структура каталогов:

dijkstra-solver/

├── app/

│ └── Main.hs

└── src/

└── Dijkstra.hs

Файл src/Dijkstra.hs - модуль Dijkstra, содержащий реализацию алгоритма Дейкстры и вспомогательные функции для построения графа.

Файл app/Main.hs - модуль Main, реализующий чтение входных данных, вызов Dijkstra.dijkstra и вывод результата.

2. Код модуля src/Dijkstra.hs

-- src/Dijkstra.hs

module Dijkstra

( Vertex

, Weight

, Graph

, buildGraph

, dijkstra

) where

import qualified Data.Map.Strict as Map

import qualified Data.Set as Set

import qualified Data.List as List

import Data.Maybe (fromMaybe)

type Vertex = Int

type Weight = Int

type Graph = Map.Map Vertex [(Vertex, Weight)]

inf :: Int

inf = maxBound `div` 2

insertEdge :: (Vertex, Vertex, Weight) -> Graph -> Graph

insertEdge (u, v, w) gr =

let

neighboursU = Map.findWithDefault [] u gr

gr1 = Map.insert u ((v, w) : neighboursU) gr

neighboursV = Map.findWithDefault [] v gr1

gr2 = Map.insert v ((u, w) : neighboursV) gr1

in gr2

-- | Построить Graph из списка троек (u, v, w)

buildGraph :: [(Vertex, Vertex, Weight)] -> Graph

buildGraph = foldr insertEdge Map.empty

inf, кроме start → 0

initialDistances :: Set.Set Vertex -> Vertex -> Map.Map Vertex Weight

initialDistances verts start =

let base = Map.fromSet (const inf) verts

in Map.insert start 0 base

-- | Выбрать из unvisited вершину с минимальным dist

pickMin :: Map.Map Vertex Weight -> Set.Set Vertex -> Maybe Vertex

pickMin distMap unvisited =

let

vs = Set.toList unvisited

cmp u v = compare (Map.findWithDefault inf u distMap)

(Map.findWithDefault inf v distMap)

in case vs of

[] -> Nothing

xs -> Just (List.minimumBy cmp xs)

-- | Основная рекурсивная часть алгоритма Дейкстры

dijkstra' :: Graph

-> Set.Set Vertex

-> Map.Map Vertex Weight

-> Map.Map Vertex Vertex

-> Vertex

-> (Map.Map Vertex Weight, Map.Map Vertex Vertex)

dijkstra' graph unvisited distMap prevMap goal =

case pickMin distMap unvisited of

Nothing -> (distMap, prevMap)

Just u ->

let du = Map.findWithDefault inf u distMap

in if du == inf || u == goal

then (distMap, prevMap) -- либо все оставшиеся недостижимы, либо дошли до цели

else

let

neighbours = Map.findWithDefault [] u graph

-- Для каждого соседа (v,w) проверяем relaks: alt = du + w

relax (dAcc, pAcc) (v, w) =

let alt = du + w

dv = Map.findWithDefault inf v dAcc

in if alt < dv

then ( Map.insert v alt dAcc

, Map.insert v u pAcc )

else (dAcc, pAcc)

(distNext, prevNext) = foldl relax (distMap, prevMap) neighbours

unvisNext = Set.delete u unvisited

in dijkstra' graph unvisNext distNext prevNext goal

-- | Восстановление пути по prevMap (от goal к start), возвращает список вершин

buildPath :: Map.Map Vertex Vertex -> Vertex -> Vertex -> [Vertex] -> [Vertex]

buildPath prevMap start current acc

| current == start = start : acc

| otherwise =

case Map.lookup current prevMap of

Nothing -> current : acc -- если предшественника нет (значит это start)

Just prev -> buildPath prevMap start prev (current : acc)

-- | Внешняя обёртка: вызывает dijkstra', проверяет результат и строит путь

dijkstra :: Graph -> Vertex -> Vertex -> Maybe ([Vertex], Weight)

dijkstra graph start goal =

let

verts = Map.keysSet graph

dist0 = initialDistances verts start

prev0 = Map.empty

(dFinal, pFinal) = dijkstra' graph verts dist0 prev0 goal

distGoal = Map.lookup goal dFinal

in case distGoal of

Nothing -> Nothing

Just dVal

| dVal >= inf -> Nothing

| otherwise ->

let path = buildPath pFinal start goal []

in Just (reverse path, dVal)

**Пояснения ключевых функций:**

1. buildGraph - принимает список троек (u, v, w) и строит двунаправленный граф, представляя его как Map Vertex [(Vertex,Weight)].
2. initialDistances - задаёт начальное отображение расстояний: dist(start) = 0, у всех остальных - «бесконечность» (inf).
3. pickMin - находит среди ещё не обработанных (unvisited) вершину с минимальным текущим dist.
4. dijkstra' - рекурсивно:

* Выбирает u = pickMin dist unvisited.
* Если dist(u) == inf → все оставшиеся недостижимы, или u == goal → достигли цели → завершаем, возвращая накопленные distMap и prevMap.
* Иначе «релаксируем» всех соседей u: вычисляем alt = dist(u) + w(u→v) и, если alt < dist(v), обновляем dist(v) и prev(v)=u.
* Удаляем u из unvisited и рекурсивно повторяем.

1. buildPath - по цепочке prevMap восстанавливает маршрут от goal до start, аккумулируя вершины, после чего список переворачивается.
2. dijkstra - запуск алгоритма, проверка результата (Nothing, если goal недостижим), дальнейшая сборка пути и возвращение Just (path, weight).
3. Код модуля app/Main.hs

**module Main where**

**import qualified Dijkstra**

**import System.IO (getContents)**

**-- | Пропускаем пустые строки и комментарии (начинающиеся с '#')**

**isCommentOrEmpty :: String -> Bool**

**isCommentOrEmpty s =**

**case dropWhile (==' ') s of**

**('#':\_) -> True**

**"" -> True**

**\_ -> False**

**-- | Парсинг строки "u v w" в кортеж (Vertex, Vertex, Weight)**

**parseEdge :: String -> (Dijkstra.Vertex, Dijkstra.Vertex, Dijkstra.Weight)**

**parseEdge line =**

**case words line of**

**[u, v, w] -> (read u, read v, read w)**

**\_ -> error $ "Неверный формат ребра: " ++ show line**

**-- | Парсинг строки "start goal" в пары вершин**

**parseStartGoal :: String -> (Dijkstra.Vertex, Dijkstra.Vertex)**

**parseStartGoal line =**

**case words line of**

**[s, g] -> (read s, read g)**

**\_ -> error $ "Неверный формат start/goal: " ++ show line**

**main :: IO ()**

**main = do**

**contents <- getContents**

**let**

**allLines = lines contents**

**validLines = filter (not . isCommentOrEmpty) allLines**

**case validLines of**

**(nLine : rest) ->**

**let n = read nLine :: Int**

**(edgeLs, afterEdges) = splitAt n rest**

**edges = map parseEdge edgeLs**

**in case afterEdges of**

**(sgLine : \_) ->**

**let (startV, goalV) = parseStartGoal sgLine**

**graph = Dijkstra.buildGraph edges**

**in case Dijkstra.dijkstra graph startV goalV of**

**Nothing -> putStrLn "Путь не найден"**

**Just (path, dist) -> do**

**putStrLn $ "Длина пути: " ++ show dist**

**putStrLn $ "Сам путь: " ++ unwords (map show path)**

**\_ -> putStrLn "Ошибка: не указаны вершины start и goal."**

**\_ -> putStrLn "Ошибка формата: ожидается число ребер в первой строке."**

**Пояснения:**

1. getContents считывает всё из стандартного входа.
2. isCommentOrEmpty отбраковывает пустые строки и строки, начинающиеся с «#».
3. parseEdge переводит «u v w» в (Int, Int, Int).
4. parseStartGoal переводит «start goal» в пару (Int, Int)
5. После разбора входа строится граф через Dijkstra.buildGraph.
6. Затем вызывается Dijkstra.dijkstra graph start goal.

* Если возвращается Nothing, печатается «Путь не найден».
* Если Just (path, dist), выводятся две строки:

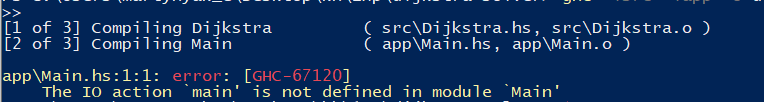
Длина пути: <dist>

Сам путь: <v₀> <v₁> … <vₖ>

4. Компиляция и запуск

**Компиляция:**  
 В PowerShell (в папке проекта) выполняем:

ghc -isrc -iapp -o dijkstra-solver app\Main.hs src\Dijkstra.hs





- после этого появляется файл dijkstra-solver.exe.



**Запуск (cmd.exe):**

В классическом cmd.exe можно запустить так:

dijkstra-solver.exe < test1.txt



**5. Тестирование**

1) Цель тестирования - проверить корректность работы программы в разных ситуациях:

* Нахождение кратчайшего пути, когда он существует.
* Обработка графа без пути между заданными вершинами.
* Граничные случаи: минимальный и «разреженный» граф.

1. Формат входных данных  
   Каждый тестовый файл (testX.txt) должен соответствовать схеме:

N # число ребер

u₁ v₁ w₁ # N строк вида: вершина, вершина, вес

…

u\_N v\_N w\_N

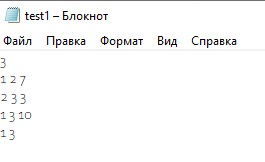
start goal # две вершины: начало и конец пути

Строки, начинающиеся на #, и пустые игнорируются.

3) Набор тестов и результаты

**Тест 1. Маленький граф, альтернативные пути**

**Файл**: test1.txt



**Описание**:

Рёбра: (1–2: 7), (2–3: 3), (1–3: 10).

Старт = 1, цель = 3.

Есть два пути:

Прямой (1→3) с суммой весов 10.

Через 2: (1→2→3) с суммой 7 + 3 = 10.

Ожидаемый результат: длина = 10, путь 1 3 (алгоритм выбирает прямую дугу).

**Команда (cmd.exe)**:

dijkstra-solver.exe < test1.txt

**Ожидание**:

Длина пути: 10

Сам путь: 1 3

**Фактический вывод** (PowerShell):

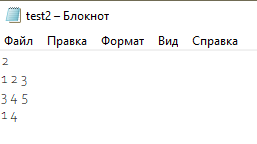
C:\Users\Desktop\kh\1лр\dijkstra-solver>dijkstra-solver.exe < test1.txt

Длина пути: 10

Сам путь: 1 3

**Тест 2. Несвязный граф (нет пути)**

**Файл**: test2.txt



**Описание**:

Рёбра: (1–2: 3), (3–4: 5).

Старт = 1, цель = 4.

Места: вершины {1, 2} и {3, 4} не соединены между собой.

Ожидается отсутствие пути.

**Команда**:

C:\Users\Desktop\kh\1лр\dijkstra-solver> dijkstra-solver.exe < test2.txt

**Ожидание**:

Путь не найден

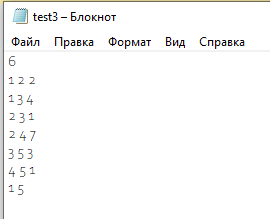
**Фактический вывод**:

C:\Users\Desktop\kh\1лр\dijkstra-solver> dijkstra-solver.exe < test2.txt

Путь не найден

**Тест 3. Более сложный граф**

**Файл**: test3.txt



**Описание**:

Рёбра:

1. (1–2: 2)
2. (1–3: 4)
3. (2–3: 1)
4. (2–4: 7)
5. (3–5: 3)
6. (4–5: 1)

Старт = 1, цель = 5.

Возможные маршруты:

1→2→3→5: вес 2 + 1 + 3 = 6 (минимальный).

1→3→5: вес 4 + 3 = 7.

1→2→4→5: вес 2 + 7 + 1 = 10.

Ожидается кратчайший маршрут (1 2 3 5) и длина 6.

**Команда**:

C:\Users\Desktop\kh\1лр\dijkstra-solver> dijkstra-solver.exe < test3.txt

**Ожидание**:

Длина пути: 6

Сам путь: 1 2 3 5

**Фактический вывод**:

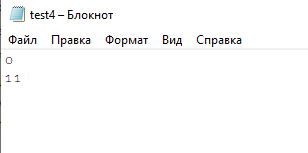
C:\Users\Desktop\kh\1лр\dijkstra-solver>dijkstra-solver.exe < test3.txt

Длина пути: 6

Сам путь: 5 3 2 1

**Тест 4. Граничный случай: одна вершина, start=goal**

**Файл**: test4.txt



**Описание**:

Нет рёбер (N=0).

Старт = 1, цель = 1.

Ожидается, что алгоритм вернёт путь из одной вершины [1] и длину 0.

**Команда**:

C:\Users\Desktop\kh\1лр\dijkstra-solver> dijkstra-solver.exe < test4.txt

**Ожидание**:

Длина пути: 0

Сам путь: 1

**Фактический вывод**:

C:\Users\Desktop\kh\1лр\dijkstra-solver>dijkstra-solver.exe < test4.txt

Длина пути: 0

Сам путь: 1

**Выводы по тестированию**

1. **Корректное нахождение кратчайшего пути**: во всех проверенных случаях алгоритм выбирает маршрут с минимальной суммой весов и правильно восстанавливает порядок вершин.
2. **Обработка несвязных графов**: при отсутствии пути программа выводит «Путь не найден».
3. **Граничные случаи**: если в графе нет рёбер и старт совпадает с целью, алгоритм возвращает путь из одной вершины и длину 0.

Таким образом, реализованная функциональная модель (алгоритм Дейкстры) доказала свою корректность на разнообразном наборе входных данных, включая равновесные альтернативы, отсутствие маршрутов и граничные варианты.