## Описание решение в общем виде.

1. Из всего списка слов отбираем для дальнейшего рассмотрения только слова "стандартной" длины (равной длине начального и конечного слов), далее буду использовать уже именно "почищенный словарь".
2. Вводим граф, у которого:
   1. вершины соответствуют словам из (почищенного) словаря - т.е. каждой вершине взаимно однозначно соответствует ровно одно слово из словаря ... и - наоборот - каждому слову из (почищенного) словаря соответствует ровно одна вершина ребра
   2. (переходы) графа "рисуются" между теми вершинами, слова которых отличаются РОВНО на одну букву

граф получается неориентированным и с одинаковыми весами переходов

1. В такой постановке задача сводится к классической задаче поиска кратчайшего пути в графе.

Можно вспомнить классический алгоритм Дейкстры поиска кратчайшего пути в графе.

Но учитывая, что граф имеет простую топологию (неориентированный, с одинаковыми весами переходов), можно сформулировать следующий алгоритм.

## Реализация алгоритма.

В алгоритме используем две следующие структуры данных:

* Каждой вершине ставится в соответствие «вес», который равен расстоянию от этой вершины до «целевой вершины». Эти веса храним в векторе Distance.
* Для каждой вершины указываем индекс вершины, через которую идет кратчайший путь к «целевой вершине». Эти индексы храним векторе Previous.
* Первоначально вектора Distance и Previous заполнены спец.значением, которое не может "появиться" в них в процессе вычислений. В качестве такого значения берется число «элементов + 1».
* Также используем два «служебных вектора» Src и Dst – их назначение объяснено ниже.
* И параметр «Длина».

В процессе исполнения алгоритма «идем» в обратном направлении – от «ЦелевойВершины» (2-ая строка) к «ИсходнойВершине» (1-ая строка).

Алгоритм выполняется следующим образом:

1. Параметру «Длина» присваиваем 0. Берем «ЦелевуюВершину», в соответствующей ей ячейке в векторе Distance, записываем значение «Длины» (на данном шаге … = 0).
2. Кладем индекс «ЦелевойВершины» в массив Src.
3. Далее выполняем следующие шаги (пока не «дойдем» до ИсходнойВершины или не переберем все вершины графа):
   1. Инкрементируем параметр «Длина».
   2. По каждому индексу из массива Src:
      1. Берем вершину (назовем ее «В»), соответствующую этому индексу (назовем его «ИндексВ»)
      2. Перебираем все вершины, к которым ведут грани из вершины «В»
      3. Отбираем вершины, для которых в ячейке массива Distance еще не был вычислен, кладем их в массив Dst.
      4. При этом как только «встречаем» индекс «ИсходнойВершины» - заканчиваем исполнение алгоритма.
      5. Для всех вершин, находящихся в Dst делаем следующие записи в векторах Distance и Previous (в ячейках этих векторов, которые соответствуют этим вершинам):
         1. В ячейки вектора Distance (которые соответствуют вершинам из Dst), заносим значение «Длина».
         2. В ячейки вектора Previous (которые соответствуют вершинам из Dst), заносим значение «ИндексВ».
   3. Переносим значения из массива Dst в массив Src. Массив Dst очищаем. Идем к шагу «а».
4. Если в процессе исполнения шага 3, мы дошли от «ЦелевойВершины» до «ИсходнойВершины», то значит путь найден. Сам путь можно «раскрутить», используя массив Previous. По факту построения полученный путь будет кратчайшим.

## Особенности проектирования.

При проектировании системы всегда надо предусматривать возможности ее последующей адаптации и улучшения. Минимально нужно предусмотреть:

1. Появление улучшенных версий алгоритмов, которые нужно интегрировать в Систему. В данном случае нужно предусмотреть возможность реализации более эффективных алгоритмов поиска кратчайшего пути в графе (алгоритм Джонсона или Левита).
2. Адаптация Системы к работе с другими форматами входной-выходной информации. В данном случае – это чтение не только из тестовых файлов, но и из бинарных, xml, etc …

Эти возможности можно реализовать разными способами.

Один из таких способов – паттерн «Стратегия».

Для данного случая полная реализация могла бы оказаться несколько тежеловатой, поэтому реализована «немного облегченная версия».

Стратегией является класс CController, в котором делается чтение-развертывание-поиск. Стратерия конкретизируется двумя классами:

* Для чтение данных - I\_SimpleFacadeOfInput
* Для обработки данных (чтения, построения графа, нахождения кратчайшего пути) - I\_SimplePathBuilder

В данном примере есть только по одной реализации:

* для чтения данных - class CTextFileInput : public I\_SimpleFacadeOfInput
* для обработчика - class CPlainPathBuilder : public I\_SimplePathBuilder

(небольшое добавление: для читателя данных из текстового файла предусмотрена возможность «подстройки» через (параметр Calibrate) – например, чтобы добавить «приведение к строчному виду»).

В данном случае для конкретизации Стратегии я использую подход с фабриками класса. CController использует для получения конкретизаций следующие фабрики классов:

* Для I\_SimpleFacadeOfInput фабрику класса FACTORY\_I\_SimpleFacadeOfInput
* Для I\_SimplePathBuilder фабрику класса FACTORY\_I\_SimplePathBuilder

Это связано с тем, что подход с фабриками классов удобен для организации юнит-тестирования. Правда, данном случае «устройство» контроллера (код его реализации) очень простое, так что тесты особо не нужны. Но архитектурный подход требует своего.

## Структура данных для хранения графа.

«По хорошему» для хранения графа хорошо бы использовать вектор списков – как это сделано, к примеру, в Boost Grap Library. Но в данном случае я решил использовать вектор векторов, поскольку граф «стабильный» (после того, как он сформирован, он уже не изменяется) и вершинах и дугах содержится минимальная информация.

## Схема документирования

Код документирован Doxygen-стиле.