МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Высшая школа киберфизических систем и управления

Работа допущена к защите
Заведующий кафедрой
_____ А.А. Ефремов
«___» _____ 2017 г.

ВЫПУСКНАЯ РАБОТА БАКАЛАВРА
НА ТЕМУ

Тема: Применение методов многопоточного программирования к алгоритмам вычисления минимального пути в графе.

Направление: 09.03.02 – Информационные системы и технологии

Выполнил:
студент гр. 43502/2 _____ Карнов А.В.
подпись, дата

Руководитель:
к. т. н., доцент _____ Хлопин С.В.
подпись, дата

Санкт-Петербург
2017

Реферат

Работа бакалавра содержит: 70 стр., 30 рис., 8 источников, 5 таблиц.

АЛГОРИТМ ДЕЙКСТРЫ, ПОИСК КРАТЧАЙШЕГО ПУТИ, ТЕОРИЯ ГРАФОВ, ПРОЦЕСС, ПОТОК, МНОГОПОТОЧНОСТЬ, АНАЛИЗ ЭФФЕКТИВНОСТИ АЛГОРИТМОВ.

Основной целью данной работы является теоретическое и практическое изучение многопоточного программирования на примере реализации алгоритма поиска кратчайшего пути в графе — алгоритма Дейкстры. Как результат, алгоритм был реализован в двух видах однопоточном и многопоточном. Осуществлено сравнение видов реализации данного алгоритма.

В первой главе раскрываются теоретические основы многопоточного подхода реализации алгоритмов, приводятся базовые сведения из теории графов, обосновывается важность алгоритма поиска кратчайшего пути в графе, математически доказывается его корректность, во второй главе описывается анализ и выбор компонентов для реализации алгоритмов двух видов, корректность их работы тестируется. В третьей части производится сравнительный анализ эффективности реализаций, делается вывод о случаях уместного использования того или иного вида алгоритма.

Оглавление

Введение

Задача поиска кратчайшего пути (задача о минимальном пути, задача о дилижансе) — задача поиска самого короткого пути между двумя точками (вершинами) на графе, в которой минимизируется сумма весов рёбер, составляющих путь.

Задача о кратчайшем пути является одной из важнейших классических задач теории графов. Ее значимость определяется различными практическими применениями. Алгоритмы решения этой задачи являются неотъемлемой частью программного обеспечения современных GPS-навигаторов, различных ГИС, которые применяются во многих отраслях науки и техники.

С появлением многопроцессорных систем стало ясно, что многие широко применяемые алгоритмы можно сделать ещё эффективнее за счёт грамотной организации параллельного выполнения частей алгоритма.

Цель данной работы разработать многопоточный алгоритм решения задачи поиска кратчайшего пути в графе, который будет эффективней своего классического аналога.

Постановка задачи

Для того, чтобы научиться применять такой подход к написанию программ, как многопоточное программирование, требуется понять его преимущества и недостатки на примере реализации нетривиального алгоритма (например, такого, как алгоритм поиска кратчайшего пути в графе).

Для этого нужно:

- Изучить основы теории графов
- Изучить концепцию многопоточного программирования
- Понять сущность алгоритма
- На основе математического обоснования реализовать однопоточный алгоритм
- Реализовать многопоточную версию
- Проверить правильность работы как однопоточной многопоточной реализаций
- Сравнить эффективность выполнения алгоритмов.
- Сделать вывод о преимуществах и недостатках многопоточного подхода, исходя и практического опыта.

Требования к реализациям алгоритма

Под правильные реализации алгоритма будет пониматься такая реализация алгоритма, которая обладает следующими свойствами:

1. Правильность работы алгоритма доказана математически
2. Корректность работы (на наборе одних и тех же входных данных алгоритм должен демонстрировать один и тот же результат)
3. Минимальная ресурсоёмкость при максимальной производительности.

1.Теоритическая часть

1.1. Основные понятия теории графов

Граф - это совокупность непустого множества объектов - вершин и связей между ними. Объекты представляются как вершины, или узлы графа, а связи - как дуги, или рёбра.

Строгое математическое определение гласит: графом называется такая пара множеств $G = (V, E)$, где $E \subseteq [V]^2$, где E – произвольное подмножество множества $[V]^2$. Элементы множества V называются вершинами графа G, а элементы E – его рёбрами (см. Рис. 1.1).

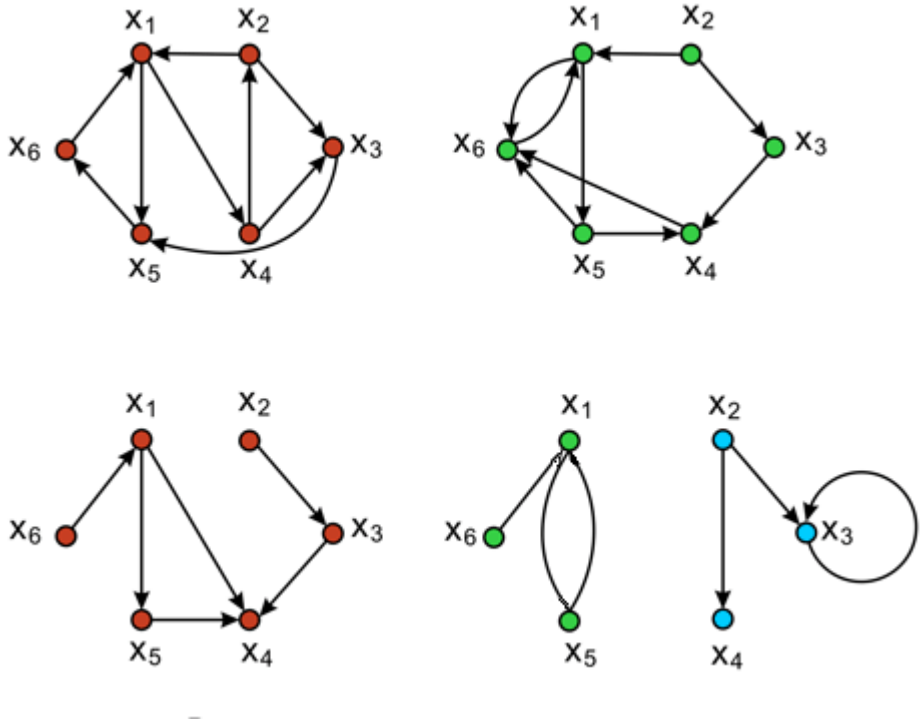

Рис №1.1 - Примеры графов

Смежными называют вершины, которые прилегают к одному и тому же ребру,

Петля это такая дуга, начальная и конечная вершина которой совпадают.

Простой граф — это граф без кратных рёбер и петель.

Степень вершины — это удвоенное количество петель, находящихся у этой вершины плюс количество остальных прилегающих к ней рёбер. Пустым называется графом называется граф без рёбер.

Полным называется такой граф, в котором каждые две вершины смежные.

Путь в ориентированном графе — это такая последовательность дуг, в которой конечная вершина всякой дуги, отличной от последней, является начальной вершиной следующей.

Если ребра ориентированы, это указывается стрелками. Они называются дугами, а граф называется ориентированным графом. Если ребра не имеют направлений, граф называется неориентированным.

1.1.1. Виды графов

В зависимости от своей конфигурации (правил взаимных расположений вершин и рёбер) выделяют следующие основные виды графов:

Граф без дуг (то есть неориентированный), без петель и кратных рёбер называется обыкновенным (см. Рис. 1.2).

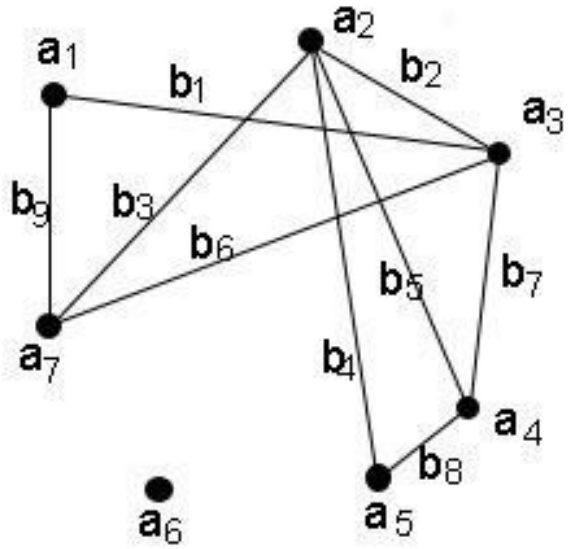

Рис №1.2 - Граф обыкновенный

Граф называется двудольным, если множество его вершин возможно разбить на два подмножества так, чтобы никакое из ребер не соединяло вершины одного и того же подмножества.

Граф заданного типа называют полным, если он содержит все возможные для этого типа рёбра (при неизменном множестве вершин). Так, в полном обыкновенном графе каждая пара различных вершин соединена ровно одним звеном (см. Рис. 1.3).

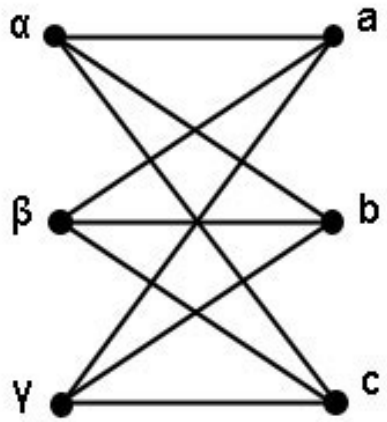

Рис №1.3 «Полный двудольный граф»

Эйлеровым графом (графом Эйлера) называется такой граф, в котором можно обойти все вершины и при этом пройти одно ребро только один раз. В нём каждая вершина должна иметь только чётное число рёбер.

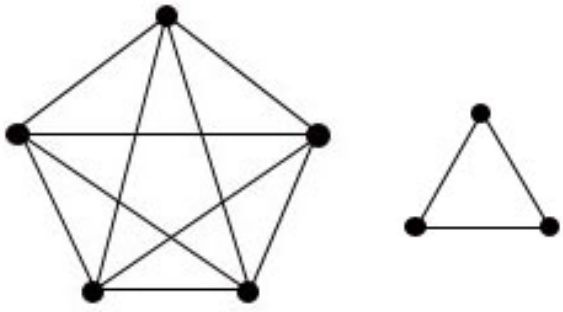

Рис №1.4 - Эйлеровы графы

Регулярным графом называется граф, все вершины которого имеют одинаковую степень. Таким образом, на Рис. 1.4 изображены примеры регулярных графов, называемых по степени его вершин 4-регулярными и 2-регулярными графами.

Гамильтоновым графом называется граф, содержащий гамильтонов цикл. Гамильтоновым циклом называется простой цикл, проходящий через все вершины рассматриваемого графа. Таким образом, гамильтонов граф - это такой граф, в котором можно обойти все вершины, и каждая вершина при обходе повторяется лишь один раз (см. Рис. 1.5).

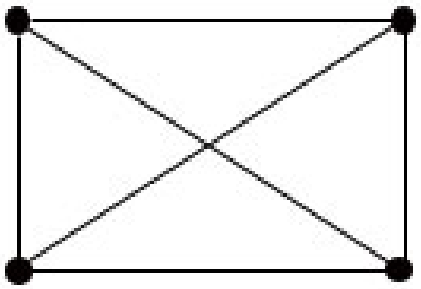

Рис №1.5 - Гамильтонов граф

Деревом называется связный граф без циклов. Связность означает наличие путей между любой парой вершин, ацикличность — отсутствие циклов и то, что между парами вершин имеется только по одному пути (см. Рис. 1.6).

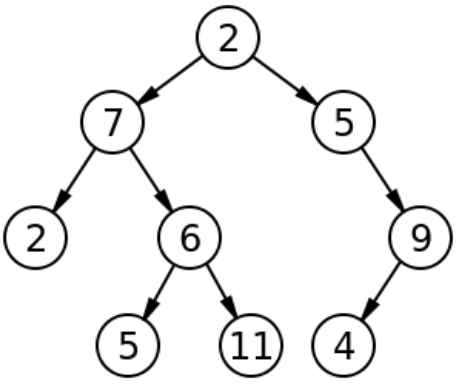

Рис №1.6 - Дерево (ациклический граф)

1.1.2. Операции на графах

Операции над графами - это такие действия над вершинами или рёбрами графов, после которых из исходного графа получается новый граф. Операции можно разделить на унарные и бинарные.

- Унарная (одноместная) операция создаёт новый граф из старого.

К основным операциям такого рода можно отнести добавление или удаление вершины, или рёбер, слияние вершин.

Пусть $G = [R, A]$ граф, где A - множество рёбер, R - множество вершин, тогда

Удалением ребра A_1 графа $G = [R, A]$ называется операция исключения A_1из множества A: $A = A \setminus A_1$

Добавлением ребра A_1 в граф $G = [R, A]$ называется операция добавления A_1к множеству множества A:$A = A \cup A_1$ при условнии, что $A_1 \nsubseteq A$

Удалением вершины R_1 графа $G = [R, A]$ называется операция исключения R_1из множества R:$R = R \setminus R_1$

Добавление вершины R_1 в граф $G = [R, A]$ называется операция добавления R_1к множеству множества R: $R = R \cup R_1$ при условии, что $R_1 \nsubseteq R$

Слиянием вершин графа$G = [R, A]$ называется операция исключения A_1из множества A: $A_1 = (A \setminus A_1) \cup A_n$, где A_n- уже существующая в графе вершина.

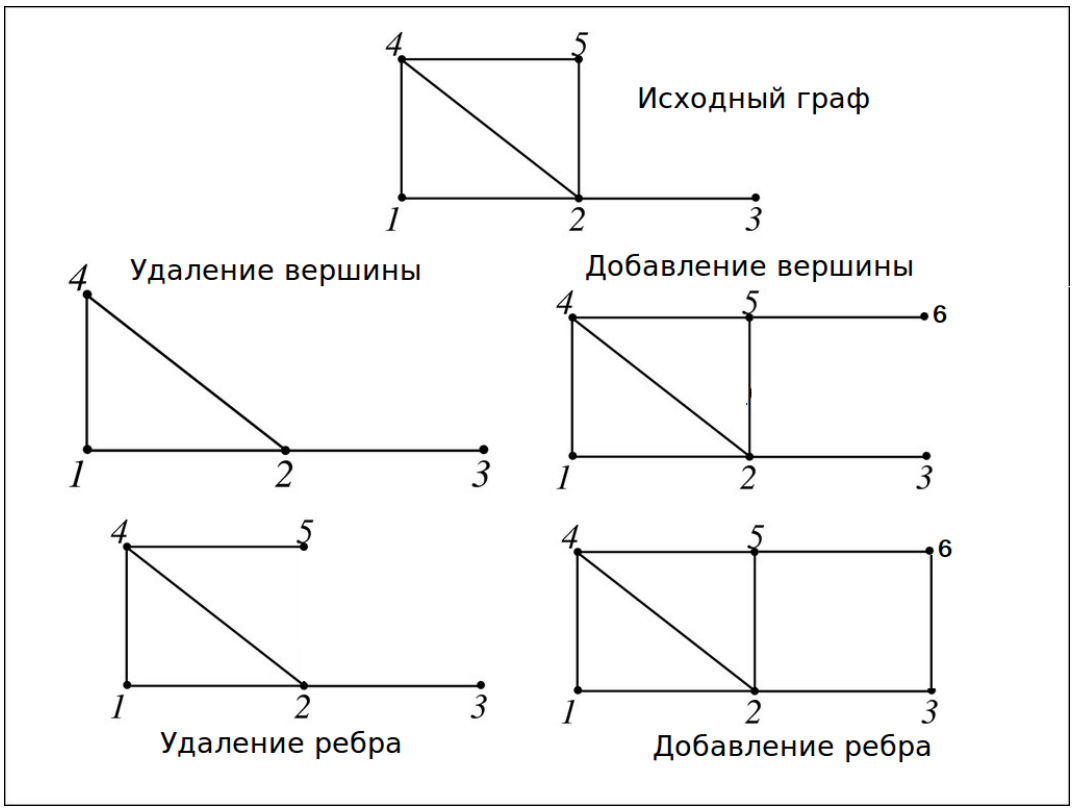

Рис №1.7 - Примеры унарных операций над графами

• Бинарная (двуместная) операция создаёт новый граф из двух исходных графов.

К основным операциям такого рода можно отнести объединение, пересечение и кольцевую сумму.

Пусть $G_1 = [R_1, A_1]$ и $G_2 = [R_2, A_2]$ два графа, где A_1, A_2 - множество рёбер, R_1, R_2 - множество вершин первого и второго графа соответственно, тогда

Объединением графов называется граф $G = [G_1 \cup G_2]$, множество вершин которого есть объединение множеств вершин графов G_1и $G_2 R = [R_1 \cup R_2]$, а множество рёбер является объединением множеств ребер этих графов $A = [A_1 \cup A_2]$.

Пересечением графов G_1и G_2называется граф $G = [G_1 \cap G_2]$, множество вершин которого $R = [R_1 \cap R_2]$, а множество ребер $A = [A_1 \cap A_2]$.

Кольцевой суммой графов G_1и G_2 называется граф$G = [G_1 \otimes G_2]$, порождённый на множестве рёбер $[A_1 \cup A_2] \setminus [A_1 \cap A_2]$, т. е. на множестве рёбер,

присутствующих либо G_1, либо в G_2, но не принадлежащих их пересечению $G_1 \cap G_2$

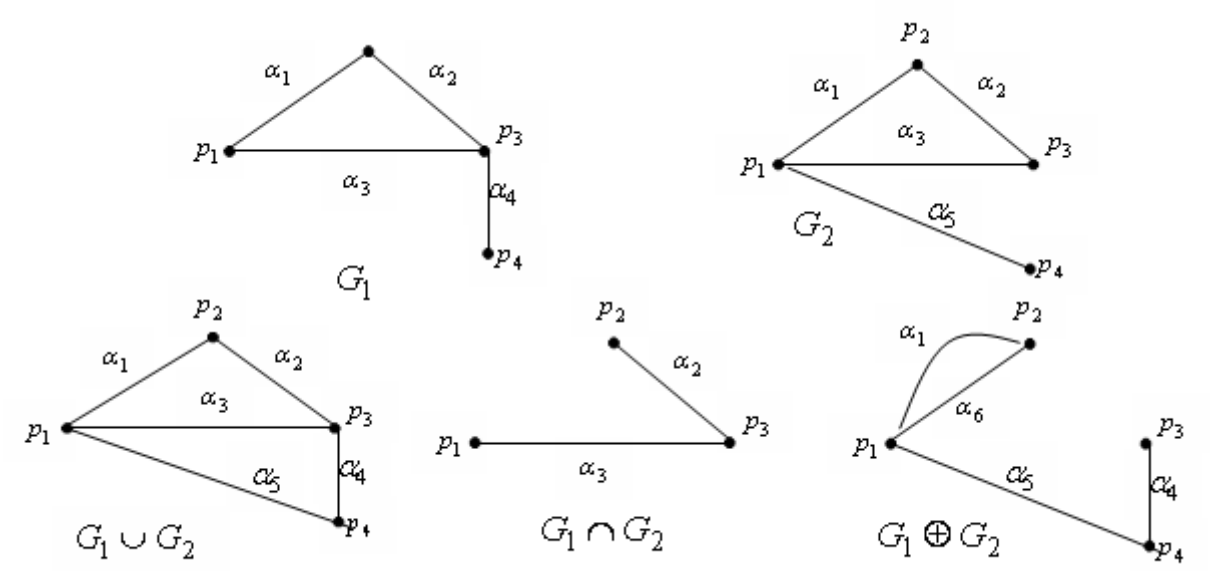

Рис №1.8 - Примеры бинарных операции над графами

1.2. Важнейшие задачи, решаемые в теории графов

Как было показано выше, теория графов имеет большой математический аппарат, который позволяет решать решить большой спектр проблем реального мира. Среди таких проблем можно выделить классические задачи теории графов — задачи, которые с решаются в комбинаторике, оптимизации уже давно и задачи специального назначения — задачи которые используются для написания программ на высокоуровневых языках программирования.

1.2.1. Классические задачи

Примерами такого рода задач являются:

Проблема семи мостов Кёнигсберга — можно рассматривать как задачу поиска оптимального пути. Решение широко применится в современной логистике для оптимизации расходов ресурсов для регулярных маршрутов.

Задача коммивояжёра — задача комбинаторной оптимизации, которая позволят отыскать оптимальный маршрут. Результаты решения этой задачи

применяются повсеместно, начиная от расчёта маршрутов в GPS — навигаторах, заканчивая оптимизации гидротехнических сооружений.

Решение таких задачи, как задача о клике, нахождение минимального остовного дерева применяются в химии (для описания структур, путей сложных реакций), схемотехнике (топология меж соединений элементов на печатной плате или микросхеме представляет собой граф).

Существуют многочисленный ряд задач, решения которых лежать в основе методов вычислительной математики. Такие понятия как изоморфизм графов, планарность графов используются для построения алгоритмов обработки видео информации.

1.2.2. Задачи специального назначения

Наряду с классическими задачами, благодаря бурному развитию вычислительной техники, стали появляться новые задачи, для решения которых удобно применять инструментарий теории графов.

Так для хранения данных и быстрого доступа к ним в информатике используют абстракции, которые называются структурами данных. Такие структуры данных как связанные списки, двоичные деревья поиска пирамиды, представляют собой ничто иное как графовые структуры. Благодаря такому подходу, операции поиска, сортировки и хранения данных в компьютерных системах имеют высокую эффективность.

Существуют криптографические методы, работа которых основана на вычислении контрольных сумм, которые в свою очередь основаны на задачах теории графов.

1.3. Многопоточное программирование

Многопоточность — это свойство платформы (например, операционной системы) или программы, которое состоит в том, что процесс, который был порождён системой, может состоять из нескольких потоков, выполняющихся

параллельно (или же псевдопараллельно), то есть без предписанного временного порядка. При выполнении некоторых задач такое разделение позволяет достичь более эффективного использования ресурсов вычислительной системы.

Основными понятия многопоточности являются понятия потока и процесса:

Процесс — это программа, которая выполняется в текущий момент.

Поток выполнения — это такая наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы.

Компьютерная программа — это пассивная последовательность инструкций, в то время как процесс — это непосредственное выполнение этих инструкций.

Процесс состоит из трёх основных компонент:

- Исполняемого кода
- Ассоциируемых с процессом данных, которые необходимы для выполнения этой программы
- Контекста – служебной информация для ОС, необходимой для управления процессом.

Поток – единица активности операционной системы, создаваемая при запуске процесса (т. е. является производной процесса) системой или программно из другого потока того же процесса.

Реализация потоков выполнения и процессов в разных операционных системах может отличается друг от друга, однако в большинстве реальных случаев поток выполнения находится внутри процесса. Несколько потоков выполнения могут существовать в рамках одного и того же процесса и совместно использовать ресурсы, например такие, как память, в то время как процессы не разделяют этих ресурсов. В частности, потоки выполнения

разделяют инструкции процесса (его код) и его контекст (значения переменных, которые они имеют в любой момент времени).

Жизненный цикл потока представляет собой запуск (зарождение), прохождение определённой последовательности выполнения задач и завершения. В каждом потоке существует указатель команд, который позволяет следить за тем, где в настоящее время происходит его выполнение в рамках текущего контекста процесса. Поток может быть прерван или переведён на время в состояние ожидания (приостановка) в то время как другие потоки продолжают работать (возврат управления).

1.3.1 Преимущества многопоточности

Многопоточность, как широко распространённая модель программирования и исполнения кода, позволяет нескольким потокам выполняться в рамках одного процесса. Эти потоки выполнения совместно используют ресурсы процесса, но могут работать и самостоятельно. Многопоточная модель программирования предоставляет разработчикам удобную абстракцию параллельного выполнения. Однако, пожалуй, наиболее интересное применение технологии имеется в том случае, когда она применяется к одному процессу, что позволяет его параллельное выполнение на многопроцессорной системе.

Это преимущество многопоточной программы позволяет ей работать быстрее на компьютерных системах, которые имеют несколько процессоров, процессор с несколькими ядрами или на кластере машин — из-за того, что потоки выполнения программ естественным образом поддаются действительно параллельному выполнению процессов.

К неоспоримым преимуществам многопоточного программирования можно отнести:

•Улучшенная реакция приложения - любая программа, содержащая много не зависящих друг от друга действий, может быть перепроектирована так, чтобы каждое действие выполнялось в отдельном потоке. Например,

пользователь многопоточного интерфейса не должен ждать завершения одной задачи, чтобы начать выполнение другой.

•Более эффективное использование мультипроцессирования - как правило, приложения, реализующие параллелизм через потоки, не должны учитывать число доступных процессоров. Производительность приложения равномерно увеличивается при наличии дополнительных процессоров. Численные алгоритмы и приложения с высокой степенью параллелизма, например, перемножение матриц, могут выполняться намного быстрее.

•Улучшенная структура программы - некоторые программы более эффективно представляются в виде нескольких независимых или полуавтономных единиц, чем в виде единой монолитной программы. Многопоточные программы легче адаптировать к изменениям требований пользователя.

•Эффективное использование ресурсов системы - Программы, использующие два или более процессов, которые имеют доступ к общим данным через разделяемую память, содержат более одного потока управления. При этом каждый процесс имеет полное адресное пространство и состояние в операционной системе. Стоимость создания и поддержания большого количества служебной информации делает каждый процесс более затратным, чем поток. Кроме того, разделение работы между процессами может потребовать от программиста значительных усилий, чтобы обеспечить связь между потоками в различных процессах или синхронизировать их действия.

1.3.2 Недостатки многопоточности

Для увеличения производительности и уменьшения количества используемых ресурсов системы рекомендуется использовать как можно меньше потоков. Кроме того, при использовании потоков необходимо учитывать требования к ресурсам и возможность возникновения ошибок при разработке приложения. Существуют следующие требования к ресурсам.

- Память, используемая системой для обработки контекстных данных, необходимых в процессах. Поэтому количество создаваемых процессов, объектов и потоков ограничено объемом доступной памяти.
- Время процессора, используемое для отслеживания количества потоков. Если количество потоков велико, большинство из них не будет работать должным образом. Если большинство текущих потоков находится в одном процессе, потоки других процессов планируются реже.
- Контроль выполнения кода с большим количеством потоков достаточно сложен и может являться причиной ошибок.

При совместном доступе к ресурсам могут возникать конфликты. Для их предотвращения нужно синхронизировать или контролировать доступ к общим ресурсам. Ошибка синхронизации доступа (в одном или разных доменах приложения) может привести к взаимоблокировке (поток перестает выполняться, ожидая завершения работы другого потока, который, в свою очередь, ожидает завершения работы первого потока) и состояниям гонки (возникновение непредвиденных результатов из-за неверной временной зависимости между двумя событиями). Система содержит синхронизирующие объекты, используемые для управления распределением ресурсов между потоками.

Ресурсы, требующие синхронизации:

- системные ресурсы (последовательные порты);
- ресурсы, используемые несколькими процессами (дескрипторы файлов);
- ресурсы отдельного домена приложения (глобальные, статические поля или поля экземпляров), к которым обращаются несколько потоков.

1.3.3 Использование многопоточности

Как известно, потоки иногда называют легковесными процессами, однако фундаментальным отличием потоков и процессов является то, что память различных процессов отделена, а потоки работают в одном и том же

адресном пространстве. Хотя последнее обстоятельство и упрощает использование совместных ресурсов потоками, но также и делает более вероятным ошибку, когда один поток меняет данные другого, поэтому при многопоточном программировании рекомендуется использовать объекты синхронизации, такие как мютексы и критические секции. При правильном использовании многопоточность позволяет программисту провести декомпозицию пользовательского интерфейса и «реальной работы».

Почти при любом использовании потоков необходимо уметь разделять данные между потоками. Когда два потока пытаются получить доступ к одним данным (не зависимо от того являются ли эти данные объектом или ресурсами), необходимо синхронизировать доступ к ним, чтобы избежать доступа или изменения данных более чем одним потоком одновременно. Всегда существуют так называемые инварианты в предположения программы о соответствующих элементах данных. Например, для списка таким предположением является то, что указатель указывает на первый элемент, а каждый элемент содержит указатель на следующий или NULL в случае последнего элемента. Во время вставки нового элемента в сисок существует момент, когда этот инвариант нарушен. Если список используется из двух потоков, то вы должны защитить этот момент, чтобы в этот момент список больше никем не использовался. Очень важно убедится, что разделяемые данные не только захватываются потоком, но и что другие потоки не могут в этот момент работать с ними.

Важным понятием в многопоточном программировании является мьютекс. Имя данного объекта произошло от "mutual exclusion". Объект представляет собой простейший в использовании объект синхронизации. Он позволяет убедиться, что только один поток работает с заданным куском данных.

Одной из главных проблем многопоточного программирования является взаимная блокировка потоков. Взаимная блокировка возникает, когда два потока ждут освобождения ресурса, которые другой поток уже занял.

Предположим поток А уже захватил мютекс 1 и поток Б захватил мютекс 2. Если поток А ждет освобождения мютекса 2, а Б — освобождения 1, то они будут ждать вечно.

Некоторые ОС способны распознать такие случаи и возвратить специальный код ошибки, например, MUTEX_DEAD_LOCK при вызове методов Lock, Unlock или TryLock. В остальных системах программа просто зависает, пока пользователь вручную не завершит ее. Существует два стандартных пути решения данной проблемы:

1. Использование фиксированного порядка. Строится постоянная иерархия захвата объектов. Для предыдущего примера каждый поток обязан сначала захватить мютекс 1, а только потом захватить мютекс 2. В этом случае взаимная блокировка произойти не сможет.

2. Использование тестового захвата. Захватить первый объект, а далее вызвать TryLock для захвата следующего объекта. Если вызов закончился неудачей освободить все ресурсы и начать захват заново. Это более трудоемкий путь, но вы можете его использовать, если использование первого метода кажется вам недостаточно гибким.

Критические секции существуют для защиты некоторого участка кода, но на практике они очень похожи на мютексы. Единственная разница в том, что мютексы обычно видны за пределами приложения и могут разделяться между процессами, а критические секции видны только внутри приложения. Это делает их более эффективными на платформах, на которых они имеют родную реализацию. В соответствии с целью различается также и терминология: если мютекс может быть захвачен и освобождён, то программа может войти и выйти из критической секции.

Переменная-условие используется для ожидания некоторых изменений разделяемых данных. Например, можно иметь условие (condition), означающее, что очередь содержит некоторые данные. Разделяемые данные, находящиеся в очереди в данном примере защищаются с помощью мютекса. Вы можете решить данную проблему с помощью цикла, который блокирует

мю- текс, тестирует доступность данных в очереди и освобождает мютекс, если данных не обнаружено. Однако этот путь очень неэффективный, так как все время работает цикл, который просто ждёт момента, чтобы занять мютекс. В таких ситуациях гораздо эффективнее использовать условия, так как поток можно остановить до тех пор, пока другой поток не подаст сигнал об изменении состояния данных. Множество потоков могут ждать выполнения одного и того же условия. В этом случае возможно разбудить один или несколько потоков.

Семафор — это некоторая смесь из счётчика и мютекса. Главное его отличие от мютекса в том, что семафор может активироваться из любого потока, а не только из владельца, поэтому можно представлять семафор как счетчик без владельца. Поток, вызывающий функцию ожидания для семафора, ожидает, когда этот счётчик станет положительным, далее метод уменьшает счетчик и делает возврат. Когда метод сработает, он захватит мютекс снова, гарантируя правильную синхронизацию.

В литературе при описании многопоточности распространён пример супермаркета. Он гласит, что если дать продавцу в супермаркете на обслуживание две линии, то это не приведёт к увеличению числа обслуженных покупателей. Таким же образом многопоточность не сделает ваше приложение быстрее (по крайней мере без специального оборудования). Лишь при использовании специальных средств можно добиться увеличения производительности. Пока на одной линии продавец ждет от покупателя кредитную карточку, то он может обслуживать покупателя на другой линии, так и мультипоточность может помогать более эффективно использовать доступные ресурсы.

1.4. Алгоритм поиска кратчайшего пути в графе

1.4.1 Алгоритм Дейкстры

Алгоритм Дейкстры — алгоритм на графах, который был разработан Эдсгером Дейкстрой в 1959 году. В ориентированном взвешенном графе $G = (V, E)$, вес рёбер которого неотрицателен и определяется весовой функцией $\omega: E \to \Re$, алгоритм находит длины кратчайших путей из заданной вершины s до всех остальных. Алгоритм работает только для графов без рёбер отрицательного веса.

В алгоритме поддерживается множество вершин U, для которых уже вычислены длины кратчайших путей до них из s. На каждой итерации основного цикла выбирается вершина $u \notin U$, которой на текущий момент соответствует минимальная оценка кратчайшего пути. Вершина u добавляется в множество U и производится релаксация всех исходящих из неё рёбер.

Алгоритм Дейкстры выполняется за время $O(N^2)$ и является асимптотически быстрейшим из известных последовательных алгоритмов для данного класса задач.

1.4.1.1 Математическое описание алгоритма

Пусть задан граф $G = (V, E)$ с весами рёбер $f(e)$ и выделенной вершиной-источником u. Обозначим через $d(v)$ кратчайшее расстояние от источника u до вершины v.

Пусть уже вычислены все расстояния, не превосходящие некоторого числа r, то есть расстояния до вершин из множества $V_r = \{v \subseteq V | d(v) \leq r\}$.

Пусть $(v, \varpi) \subseteq argmin\{d(v) + f(e) | v \subseteq V, e = (v, \varpi) \subseteq E\}$, тогда $d(\varpi) = d(v) + f(e)$, и v лежит на кратчайшем пути от u к ϖ.

Величины $d^+(\varpi) = d(v) + f(e)$, где $v \subseteq V_r$, $e = (v, \varpi) \subseteq E$, называются предполагаемыми расстояниями и являются оценкой сверху для настоящих расстояний: $d(\varpi) < d^+(\varpi)$.

1.4.1.2 Доказательство корректности алгоритмам

Пусть $G = (V, E)$ - ориентированный взвешенный граф, вес рёбер которого неотрицателен, s — стартовая вершина. Тогда после выполнения алгоритма Дейкстры $d(u) = p(s, u)$ для всех u, где — $p(s, u)$ длина кратчайшего пути из вершины s в вершину u.

По индукции доказывается, что в момент посещения любой вершины u, $d(u) = p(s, u)$.

1) На первом шаге выбирается s, для неё выполнено: $d(s) = p(s, s)$

2) Пусть для n первых шагов алгоритм сработал верно и на $n + 1$ шагу выбрана вершина u. Докажем, что в этот момент $d(u) = p(s, u)$. Для начала отмечается, что для любой вершины v, всегда выполняется $d(v) \geqslant p(s, v)$ (алгоритм не может найти путь короче, чем кратчайший из всех существующих). Пусть P — кратчайший путь из s в u, v — первая непосещённая вершина на P, z — предшествующая ей (следовательно, посещённая). Поскольку путь P кратчайший, его часть, ведущая из s через z в, u тоже кратчайшая, следовательно $p(s, v) = p(s, z) + \varpi(zv)$. По предположению индукции, в момент посещения вершины z выполнялось $d(z) = p(s, z)$, следовательно, вершина u тогда получила метку не больше чем $d(z) + \varpi(zv) = p(s, z) + \varpi(zv) = p(s, v)$, следовательно, $d(v) = p(s, v)$. С другой стороны, поскольку сейчас была выбрана вершина u, её метка минимальна среди непосещённых, то есть $d(u) \leqslant d(v) = p(s, v) \leqslant p(s, u)$, где второе неравенство верно из-за ранее упомянутого определения вершины v в качестве первой непосещённой вершины на P, то есть вес пути до промежуточной вершины не превосходит веса пути до конечной вершины вследствие неотрицательности весовой функции. Комбинируя это с $d(u) \geqslant p(s, u)$, имею $d(u) \geqslant p(s, u)$, что и требовалось доказать.

Поскольку алгоритм заканчивает работу, когда все вершины посещены, в этот момент $d(u) = p(s, u)$ для всех u.

1.4.2 Пример работы алгоритма

Рассмотрю работу алгоритма Дейкстры на примере графа, показанного на рис. 1.9. Допустим, что требуется найти расстояния от 1-ой вершины до всех остальных (2, 3, 4, 5, 6).

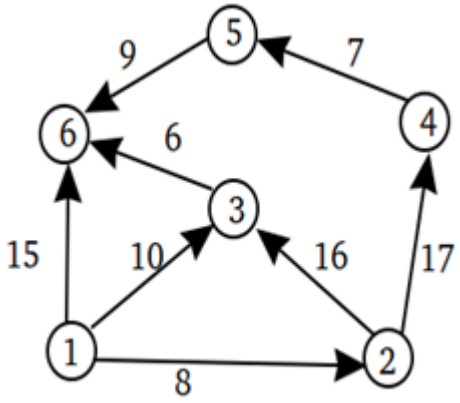

Рис №1.9 - Исходный граф

Окружностями с номерами я обозначил вершины, а стрелками — пути между ними (ребра графа). Над рёбрами обозначен их вес — длина пути между вершинами.

Красным символом бесконечности буду обозначаться ещё невычисленная длинна пути из начальной 1-ой вершины до данной (см. рис. 1.10).

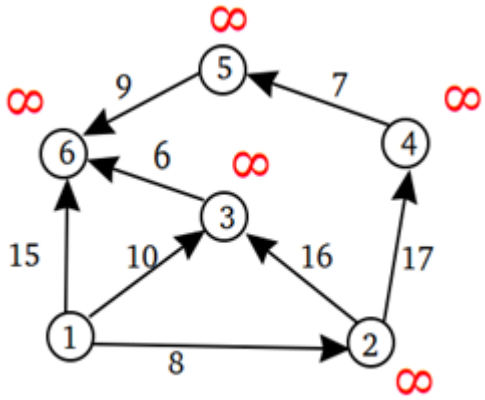

Рис №1.10 «Граф с недопрядёнными путями»

Первый шаг. Назначу опорной вершиной, от которой будет начинаться исполнение алгоритма, является вершина 1. Выделю её зелёным цветом (см. рис. 1.11). Её соседями являются вершины 2, 3 и 6.

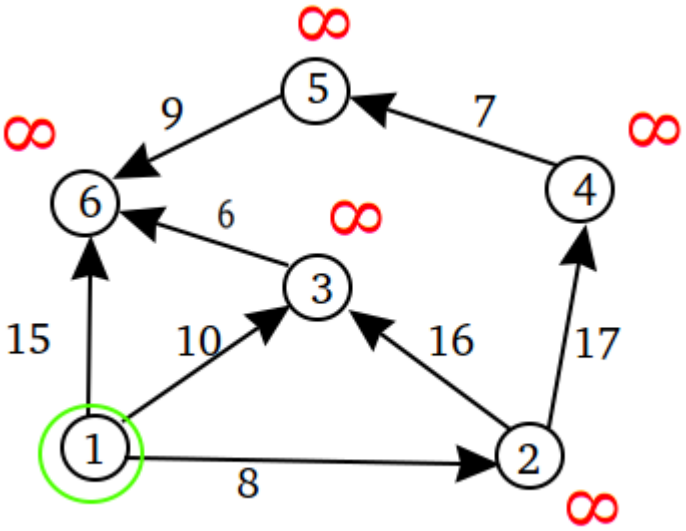

Рис №1.11 - Граф с выделенной начальной вершиной

Первый (по возрастанию индексов) сосед вершины 1 — это вершина 2, из-за того, что длина пути до неё минимальна. Длина пути через вершину 1 равна кратчайшему расстоянию до вершины 1 плюс длина ребра, которое идёт из 1 в 2, то есть $0 + 8 = 8$. Что меньше текущей метки вершины 2, поэтому новая метка 2-й вершины равна 8 (см. рис. 1.12).

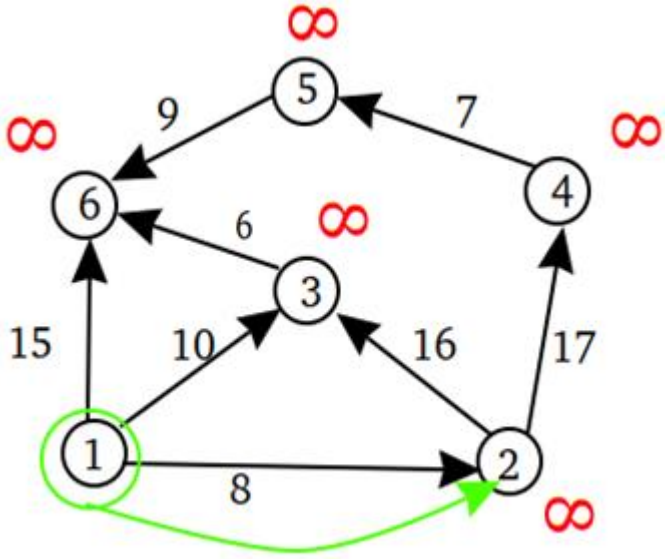

Рис №1.12 - Переход из вершины 1 к вершине 2

Аналогичную операцию буду выполнять с соседними вершинами: 1-ой, 3-ей и 6-й (см. рис. 1.13 и рис. 1.14).

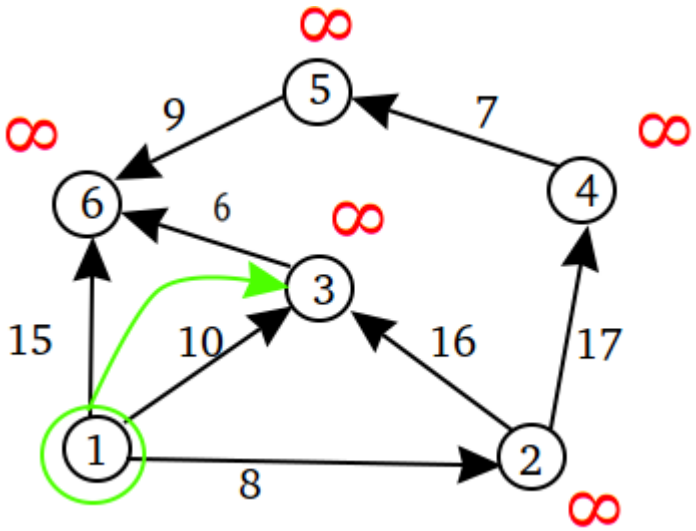

Рис №1.13 - Переход из вершины 1 к вершине 3

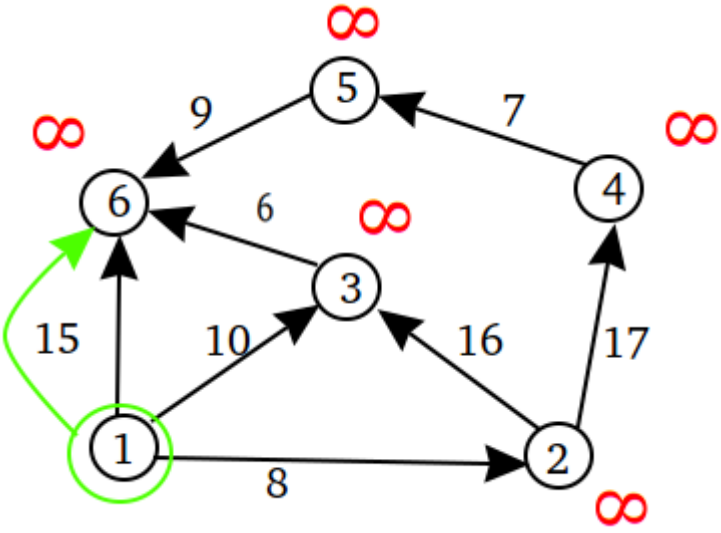

Рис №1.14 - Переход из вершины 1 к вершине 6

Теперь все соседи 1-ой вершины проверены. Минимальное расстояние до 1-ой вершины считается окончательным, не подлежащим пересмотру. Вычеркну её из графа, чтобы отметить, что эта вершина посещена (см. рис. 1.15).

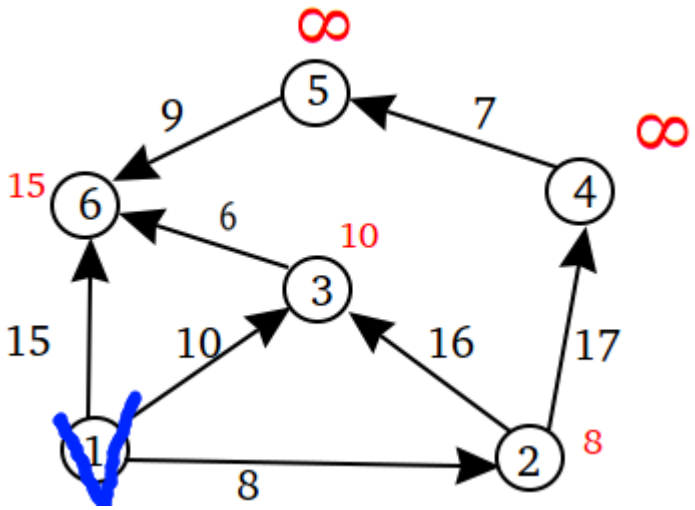

Рис №1.15 - Окончание работы с первой вершиной

Второй шаг. Теперь шаг алгоритма повторяется. Снова нахожу самую близкую из не посещённых вершин. Очевидно, что это вершина 2 с меткой 8 (см. рис. 1.16).

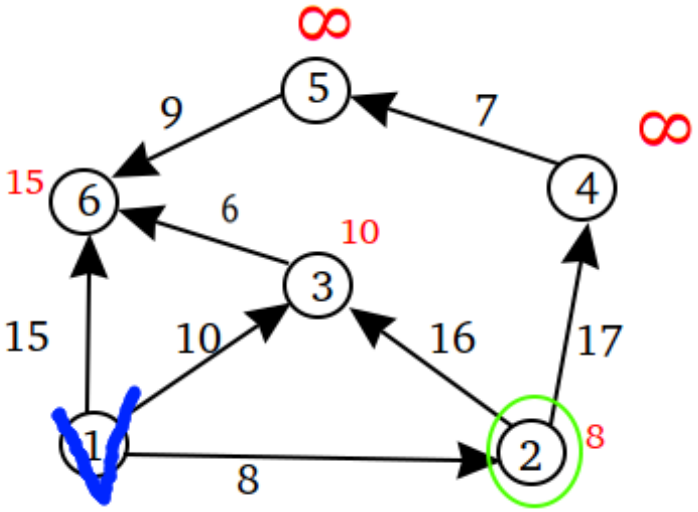

Рис №1.16 - Начало работы со второй вершиной

Теперь снова пытаюсь уменьшить метки соседей выбранной вершины, пытаясь пройти в них через 2-ую. Соседи второй это вершины под номерами 1, 3, 4.

Первый сосед вершины — это вершина 1. Одна первая вершина уже посещена, поэтому ней действий не совершается.

Следующие соседи второй вершины — это вершины 4 и 3. Если идти в неё через 2-ю, то длина такого пути будет кратчайшее расстояние до 2 -ой

вершины. Расстояние между вершинами 2 и 4 $8 + 17 = 25$. Поскольку $25 < \infty$, устанавливаю метку вершины 4 равной 25 (см. рис. 1.17).

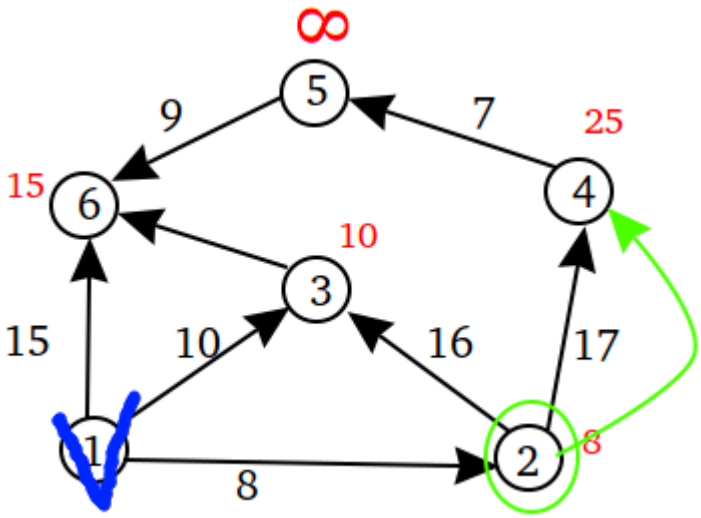

Рис №1.17 - Переход из вершины 2 к вершине 4

Ещё один сосед вершины 2 — вершина 3. Если идти в неё через 2, то длина такого пути будет $8 + 16 = 24$. Но текущая метка третьей вершины равна $10 < 24$, поэтому метка не меняется (см. рис. 1.18).

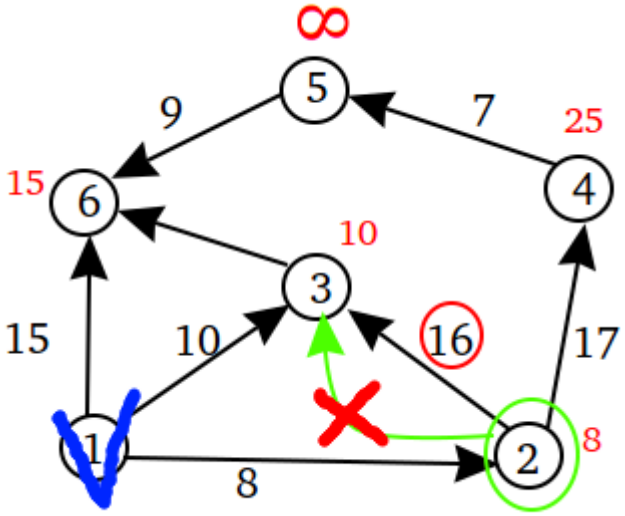

Рис №1.18 - Исключение маршрута из вершины 2 к вершине 3

Все соседи вершины 2 просмотрены, замораживаем расстояние до неё и помечаем её как посещённую.

Третий шаг. Повтор шага, при выборе 3-ей вершины.

Последующие шаги. Шаг алгоритма повторяется для оставшихся вершин (для 6-ой, 4-ой и 5-ой).

Завершение. Алгоритм заканчивает работу тогда, когда обработаны все вершины. Результат его работы - это кратчайший путь от вершины 1 до 2-й составляет 8, до 3-й — 10, до 4-й — 25, до 5-й — 31, до 6-й — 15.

1.4.3 Многопоточный алгоритм Дейкстра

Классический алгоритм Дейкстры имеет несколько очевидных недостатков, которые можно исправить, используя многопоточный подход.

- В не многопоточной реализации алгоритм не может выбрать и анализировать две вершины одновременно. Главным связующем ядром в алгоритме является цикл прохода по графу.
- Известно, что циклы плохо подаются распараллеливанию.
- При исполнении программы существует несколько точек, в которых нужно синхронизировать потоки исполнения (хранения кратчайшего расстояния до ближайшей вершины)

Главное отличие многопоточной реализации от однопоточной состоит в интерпретации и в правиле оценки вспомогательных величин , $d_i, 1 \leq i \leq n$. В классическом алгоритме Дейкстры эти величины означают суммарный вес пути от начальной вершины до всех остальных вершин графа. Как результат, после выбора очередной вершины u графа для включения в множество выбранных вершин V_t , значения величин , $d_i 1 \leq i \leq n$, пересчитываются в соответствии с новым правилом: $\forall i \notin V_t \rightarrow d_i = \min\{d_i, d_i + w(t,i)\}$. С учётом изменённого правила пересчёта величин , $d_i 1 \leq i \leq n$, схема распределения данных по процессорам при выполнении алгоритма Дейкстры отличается:

- Определяются значения величин для всех вершин, ещё не включённых в состав дерева кратчайших путей; данные вычисления

выполняются независимо на каждом процессоре в отдельности; d_i выбирается вершина t графа , имеющая дугу минимального веса до множества V.

- Для выбора такой вершины необходимо осуществить поиск минимума в наборах величин, имеющихся на каждом из процессоров (количество параллельных операций n), и выполнить сборку полученных значений.
- Рассылка номера выбранной вершины для включения в дерево кратчайших путей всем процессорам. Получение дерева кратчайших путей обеспечивается при выполнении итерации алгоритма Дейкстры.

Главный процесс выполняет следующие действия:

- разделение граф на части и осуществляет их пересылку на соответствующие потоки
- выбор стартовых вершины и рассылка и передает их на обработку всем остальным процессам
- формирование массива весов ребер с минимальным значением от любой вершины к вершине v и рассылка частей массива по соответствующим потокам

Порожденные процессы выполняют следующие действия:

- формирование минимального кратчайшего пути (добавление вершин в массив)
- прием стартовой вершины от главного процесса
- прием части массива кратчайшего пути от главного процесса
- выполнение итераций параллельного нахождения минимального поддерева дерева − определение значения величин пути для всех вершин.

2. Практическая часть

2.1. Анализ и выбор средств реализации алгоритма

Для того чтобы реализовать классический и многопоточный алгоритм Дейкстры необходимо использовать инструменты, которые поддерживают возможность реализации двух подходов. Для реализации мною были выбраны следующие инструменты:

- Язык программирования java — современный объектно-ориентированный кроссплатформенный язык программирования. Он обладает обширной стандартной библиотекой, которая позволяет без использования сторонних инструментов реализовать широкий функционал для приложения
- Junit — Фреймворк для тестирования приложений на java. Чтобы убедиться в корректности алгоритмов необходимо провести его тестирование. Возможности Junit позволят сделать это как для классического, так и для многопоточного варианта.
- Xchart - Фреймворк для построения графиков. Для демонстрации производительности приложения удобно воспользоваться графиками, которые могут быть построены данным Фреймворком.
- Maven — сборщик приложения. Воспользовавшись этим инструментом для того, чтобы собрать все компоненты приложения (скомпилированный код, Фреймворки) можно исключить появление конфликтов между частями системы.

2.2. Реализация алгоритма

Не многопоточный алгоритм поиска кратчайшего пути в графе был реализован согласно теоретическому описание (см. пункт 1.4.2). Формально его работу можно представить в виде блок-схеме (см. рис. 2.1)

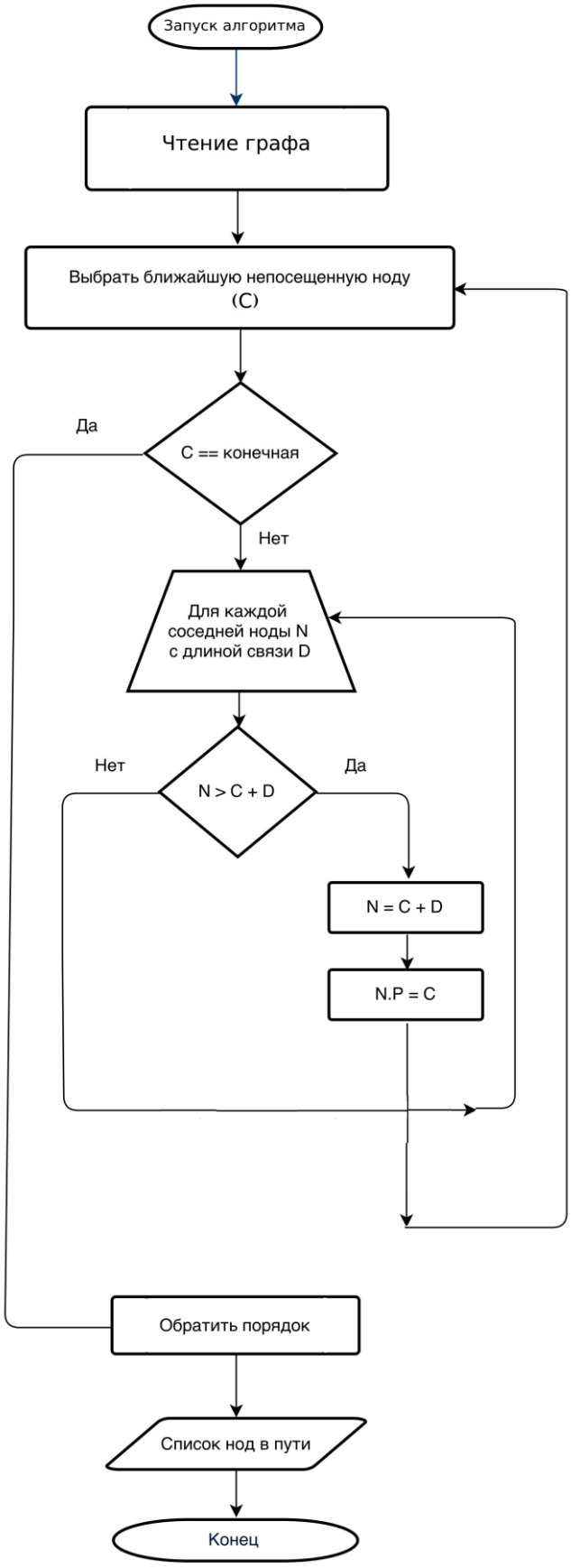

Рис №2.1 - Блок-схема алгоритма Дейкстры

Многопоточный алгоритм, несмотря на свое простое описание (см. пункт 1.4.3) сложен в технической реализации. Формально его работу можно представить в виде блок-схеме (см. рис. 2.2)

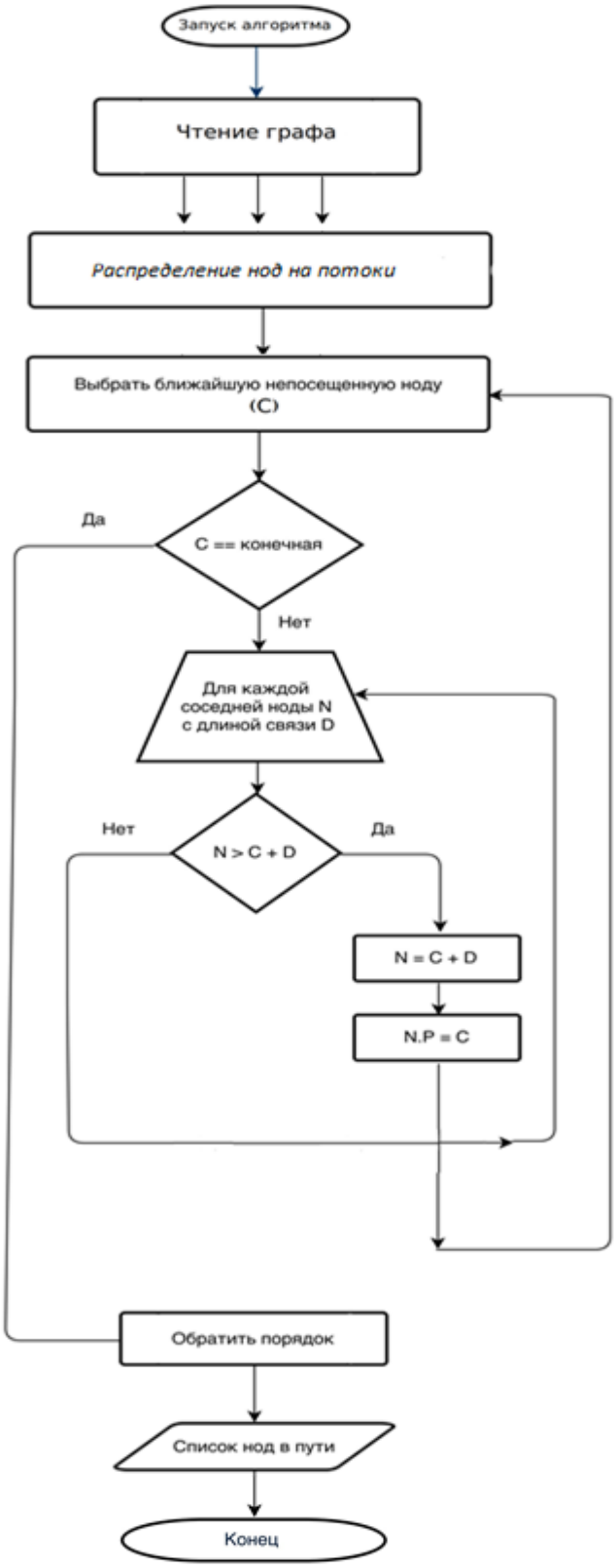

Рис №2.2 - Блок-схема параллельного алгоритма Дейкстры

Сложность многопоточной реализации заключается в том, чтобы обеспечить консистентность данных. В любом многопоточном алгоритме может возникнуть условие гонок. Для его избежание я использовал следующие инструменты, предоставляет стандартная библиотека java:

- CyclicBarrier – средство синхронизации потоков, которая обеспечивает возможность разделения работы потоков.
- BlockingQueue – структура данных, которая представляет собой очередь, дополнительно поддерживающая операции, которые ожидают, что очередь станет непустой при извлечении элемента, чтобы стать доступным в очереди при хранении элемента.
- ForkJoinPool инструмент исполнения потоков, который управляет задачами распределения ресурсов между потоками исполнения. Отличается высокой скоростью и эффективностью работы.

2.3. Проверка корректности работы алгоритма

Для тестирования алгоритмов я воспользовался Фреймворком Junit, для которого написал юнит-тесты.

Юнит-тест – это метод проверки корректности работы программного обеспечения, который заключается в том, чтобы задать входные данные – данные над которыми будут выполняться операции и выходные – данные, которые должен предоставить корректный алгоритм. После чего алгоритм запускается на входных данных, а его выходные сравниваются с ожидаемыми выходными. Такой метод тестирования позволят, когда и в какой части алгоритма возникает ошибка.

В юнит-тестах я воспользовался следующим графом.

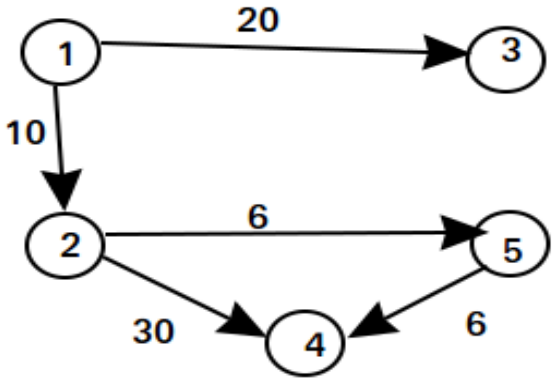

Рис №2.3 - Схема графа для тестирования

В тестах будет задано найти наикратчайший путь из вершины 1 к вершине 4. Очевидно, что результат должен выглядеть следующим образом «1→2→5→4». Выдача такого результата и будет доказательством корректной работы алгоритма.

Для однопоточного алгоритма были написаны два теста (см. код). В одном проверялось, что результат работы алгоритма из вершины 1 до вершины 4 есть «1→2→5→4», второй тест проверяет алгоритм на то, что из вершины 3 нельзя добраться в вершину 5 («Path is not real»).

Оба теста успешно завершились, тем самым доказав корректность алгоритма.

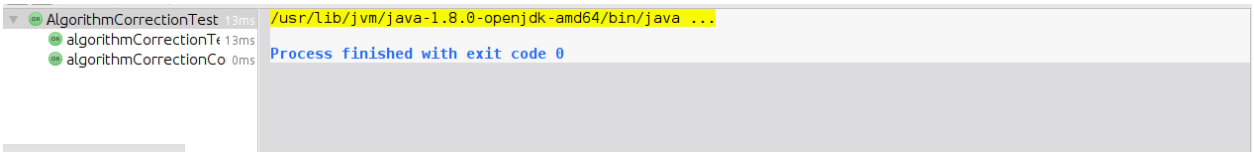

Рис №2.4 - Результат прохождения тестов классическим алгоритмом

Аналогично протестирую многопоточную реализацию. Использовав те же тесты, убеждаюсь в корректности работы и этой версии алгоритма.

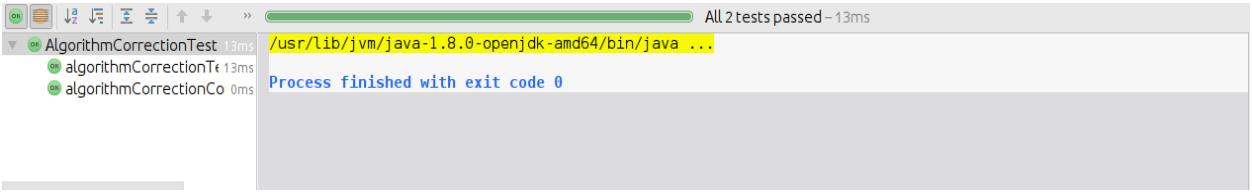

Рис №2.5 - Результат прохождения тестов многопоточным алгоритмом

Основываясь на том, что все юнит-тесты успешны пройдены классическим и многопоточный алгоритмами Дейкстры, можно сделать вывод о их корректной реализации.

3. Анализ эффективности алгоритмов

Для того чтобы проверить эффективность работы алгоритмов, были измерены две величины – время выполнения алгоритмов и количество операций, совершенных процессором за это время (загруженность процессора).

Было измерено время исполнения алгоритмов, на вход которым были предоставлены графы с 100, 1000, 10000, 100000, 1000000 вершинами.

Графы были сгенерированы по одинаковой схеме. Схема генерируется случайно, но сохраняется единой для всех тестов.

Для данной схемы случайным образом было сгенерировано распределение весов для путей. После чего схема была использована во всех последующих тестах.

Для измерения интересую я воспользовался программным комплексом JProfiler, который позволил измерить множество параметров работы алгоритмов.

Прежде всего JProfiler позволяет изолировать процесс исполнения программы и рассчитать для него параметры производительности. Например, загрузку центрального процессора, потребление оперативной памяти, активность потоков выполнения и многое другое.

На рис. 3.1 продемонстрирован процесс измерения величины нагрузки, которое производит программа на центральный процессор в процессе своего выполнения. Эти параметры можно замерит как в контексте выполнения всех компонентов операционной системы, так и изолировано (в рамках контекста java-процесса виртуальной машины).

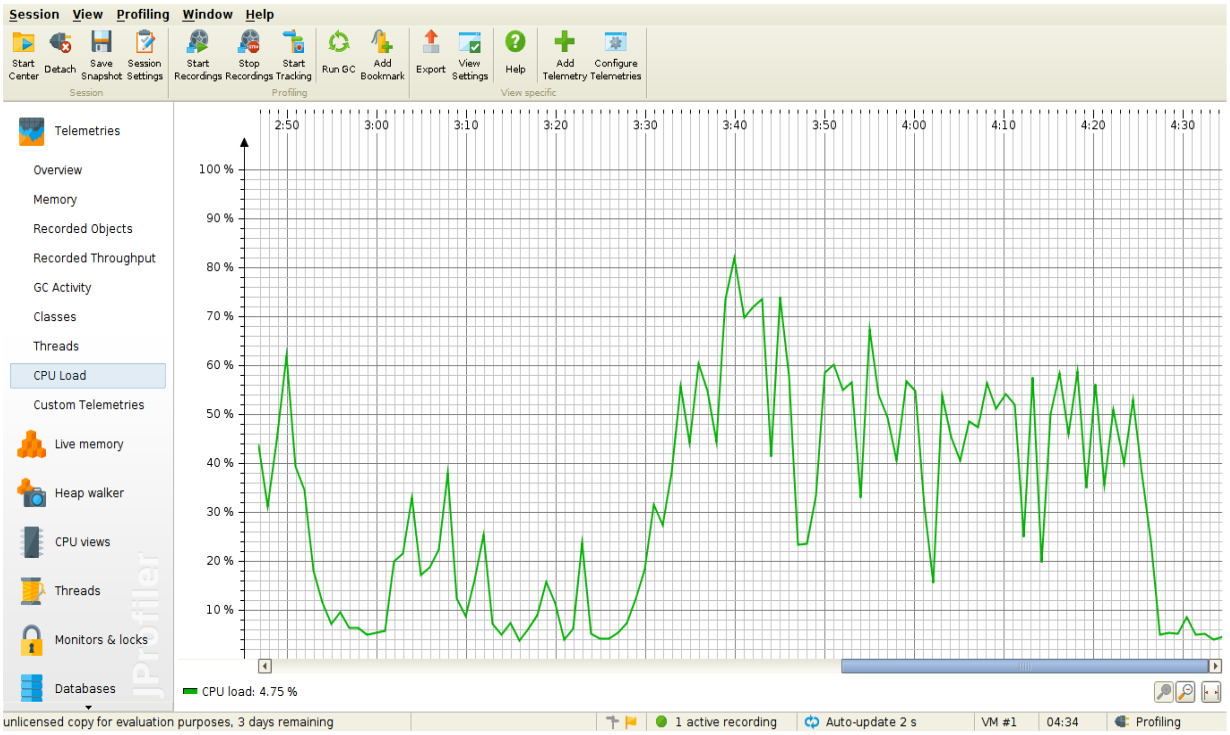

Рис №3.1 – Измерение производительности программы

На рис 3.2 продемонстрировано измерение параметров потоков выполнения программы при прохождении ей юнит-тестов.

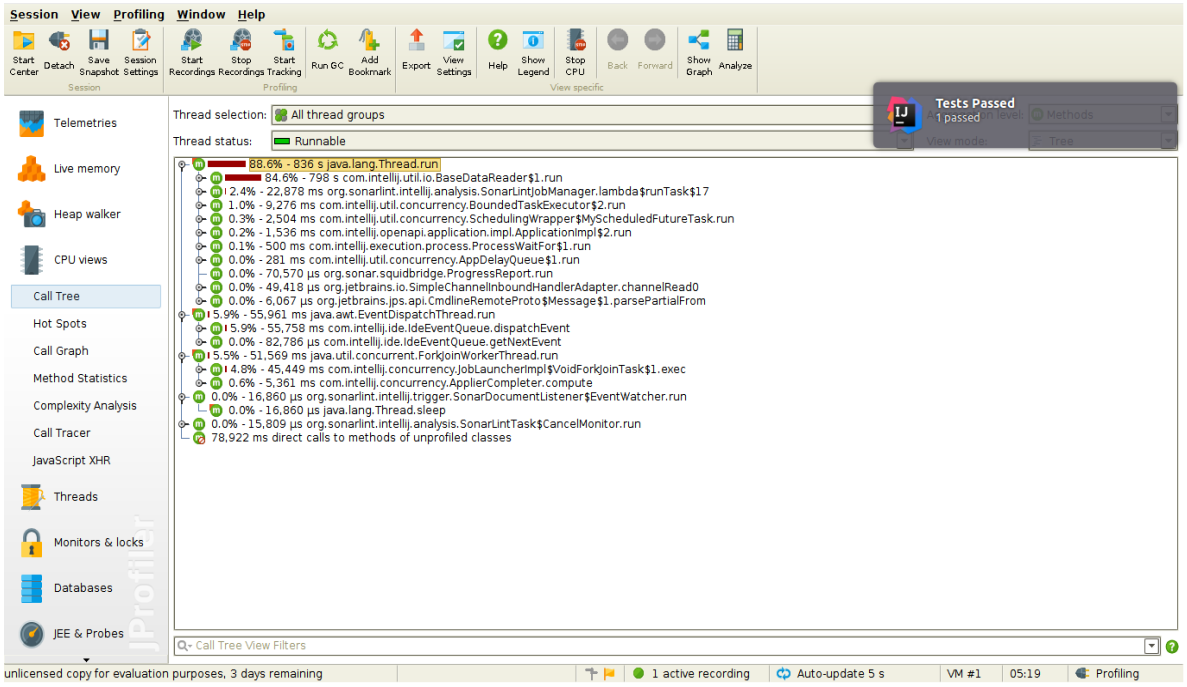

Рис №3.2 – Снятие показаний производительности потоков выполнения

Однако снятие показателей производительности – это не единственное, что можно получить от использования JProfiler. На рис. 3.3 показаны дополнительный параметры, которые возможно проанализировать.

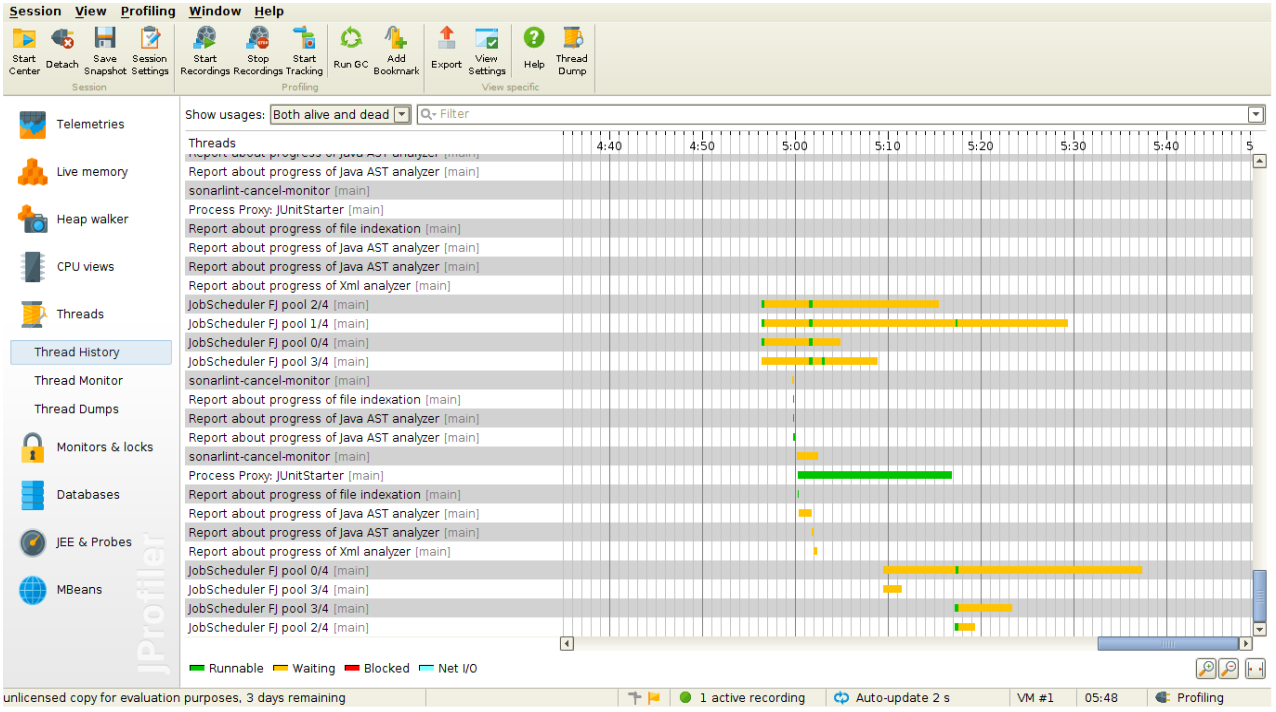

Рис №3.3 – Детальная информация о потоках выполнения

Аналогичную (дополненную) информацию можно получить о других аспектах выполнения программы (см. рис 3.4)

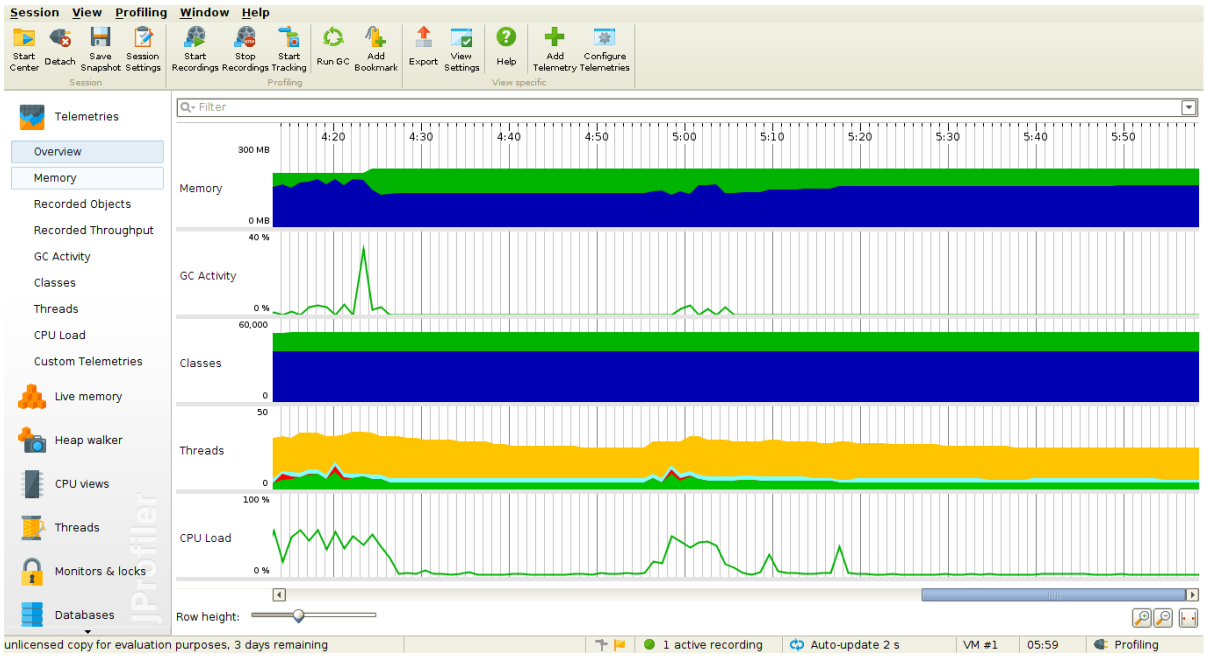

Рис №3.4 – Временная характеристика выполняющейся программы

3.1 Анализ однопоточной реализации

Для оценки эффективности реализации однопоточного алгоритма, был произведено измерение времени работы алгоритма и степени загруженности процессора при графах с 100, 1000, 10000, 100000, 1000000 вершинами, сгенерированными по схеме на рис. 3.1.

Количество вершин	Время выполнения (мкс.)	Величина загруженности процессора (в %)	Количество использованной оперативной памяти
100	21	42	13
1000	141	35	15
10000	6176	45	21
100000	95471	35	24
1000000	775532	39	27

Таблица №3.1 – Результаты измерения работы однопоточной версии алгоритма

Из данных, приведенных в таблице 3.1, очевидно, что по мере возрастания количества вершин, время исполнения алгоритма возрастало по экспоненциальной функции.

В величине за загруженности процессора не удаётся выделить общих закономерностей, так как этот показатель мог испытывать влияние.

Ожидаемо, что с ростом количества вершин, количество оперативной памяти, используемое алгоритмом, возрастает.

3.2 Анализ многопоточной реализации

Для многосторонней оценки работы многопоточного алгоритма, были произведены измерения величин, которые для программ с разным количеством потоков.

Количество вершин	Время выполнения (мкс.)	Величина загруженности процессора (в %)	Количество использованной оперативной памяти
100	53	43	21
1000	192	31	29
10000	9876	45	33
100000	212101	30	38
1000000	1995532	29	49

Таблица №3.2 – Результаты измерения работы многопоточной версии алгоритма, работающей на одном потоке исполнения

По данным таблицы 3.2 можно сделать вывод о том, что многопоточная реализация, работающая на одном потоке, проигрывает классическому алгоритму. Проигрыш происходит по всем параметрам, однако его величина измеряется не порядком, а всего лишь в 4-6 раз.

Результаты эксперимента доказывают тот факт, что для многопоточного алгоритма следует выделить насколько потоков исполнения.

Количество вершин	Время выполнения (мкс.)	Величина загруженности процессора (в %)	Количество использованной оперативной памяти
100	22	39	26
1000	152	32	35
10000	8999	33	39
100000	113133	31	45
1000000	802323	29	53

Таблица №3.3 – Результаты измерения работы многопоточной версии алгоритма, работающей на двух потоках исполнения

Данные таблицы 3.3 говорят о некотором улучшении производительности по сравнению с предыдущей реализацией.

Отставание от однопоточного алгоритма сократилось. Рассматривая время выполнения и загруженность процессора, можно сделать вывод о сторожей производительности.

Параметром, по которому эта реализация проигрывает классическому алгоритму – оперативная память, использованная программой. Однако этот факт объясняется тем, что для реализации такой структуры, как поток, необходимо достаточно большое количество оперативной памяти.

Количество вершин	Время выполнения (мкс.)	Величина загруженности процессора (в %)	Количество использованной оперативной памяти
100	20	42	32
1000	139	19	39
10000	4300	26	48
100000	71954	31	51
1000000	340978	32	63

Таблица №3.4 – Результаты измерения работы многопоточной версии алгоритма, работающей на трех потоках исполнения

Из таблицы 3.4 становится очевиден выигрыш многопоточного подхода к написанию алгоритма.

С увеличением количества вершин, многопоточный аналог практически на порядок превосходит классический.

Данный результат стал возможным благодаря достаточному количеству потоков исполнения и организации безопасного взаимодействия этих потоков через разделяемую память.

Единственным параметром, по которому данная реализация уступает многопоточному аналогу – это количество потребленной оперативной памяти. Как было упомянуто выше, это присуще всем многопоточным алгоритмам.

Количество вершин	Время выполнения (мкс.)	Величина загруженности процессора (в %)	Количество использованной оперативной памяти
100	31	42	31
1000	145	23	42
10000	7892	19	49
100000	100321	36	56
1000000	940820	38	66

Таблица №3.5 – Результаты измерения работы многопоточной версии алгоритма, работающей на четырех потоках исполнения

По данным таблицы 3.5 можно сделать вывод о том, о некотором падении (некритичном) производительности алгоритма по сравнению с однопоточной реализацией.

Видимо, синхронизация общих ресурсов стала сказываться на времени исполнения и загруженности процессора. Именно такая нетривиальная задача, как задача о избежание условия гонок отрицательно сказывается на производительности приложения.

3.3 Сравнение эффективности двух реализаций

Согласно данным приведенным в таблицах 3.1-3.5, были построены графики результатов работы алгоритмов.

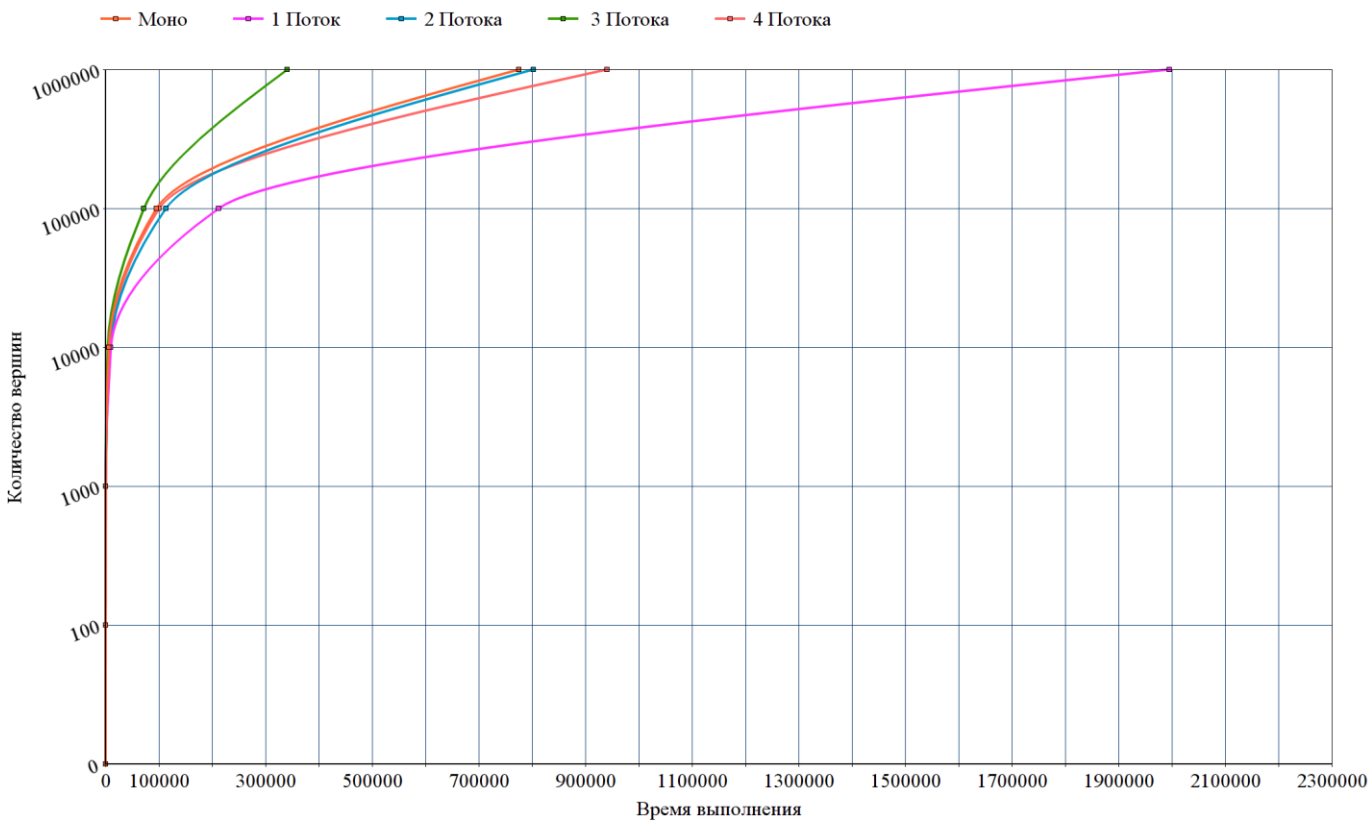

Рис №3.2 - Сравнительная диаграмма эффективности алгоритмов по времени выполнения

Самым эффективной реализаций алгоритмов по времени выполнения оказался многопоточный алгоритм, работающий с 3 потоками, а наименее эффективным – многопоточный с выделенным одним.

Выигрыш 3-ех поточной реализации объясним нахождением баланса между количеством потоков и способом синхронизации данных, с которыми они работают.

Многопоточная реализация, выполняющееся на одном потоке – самый худший вариант использования методов многопоточного программирования.

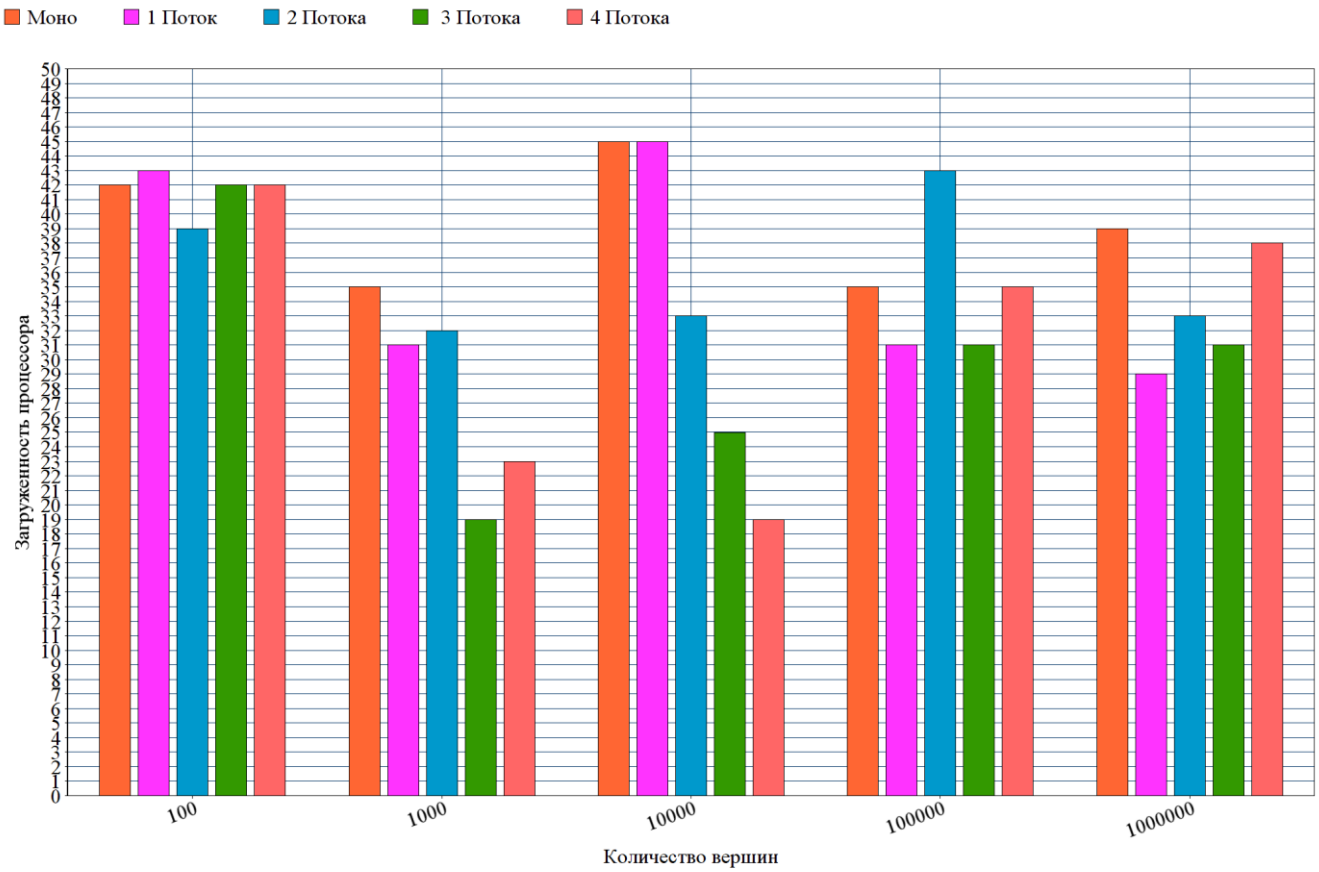

Рис №3.3 - Сравнительная диаграмма эффективности алгоритмов по загруженности процессора

Из-за того, что средство измерения JProfiler не способно измерить количество операция, произведенных процессором, была вычислена средняя загруженность процессора. Худший алгоритм по времени исполнения – многопоточный алгоритм на единственном потоке исполнения оказался таким же неэффективным и в этом тесте. Однако при большом количестве вершин графа (больше 10000) однопоточное решение оказывается хуже.

Многопоточные реализации, работающие на нескольких потоках исполнения, оказались менее требовательны к ресурсам процессора.

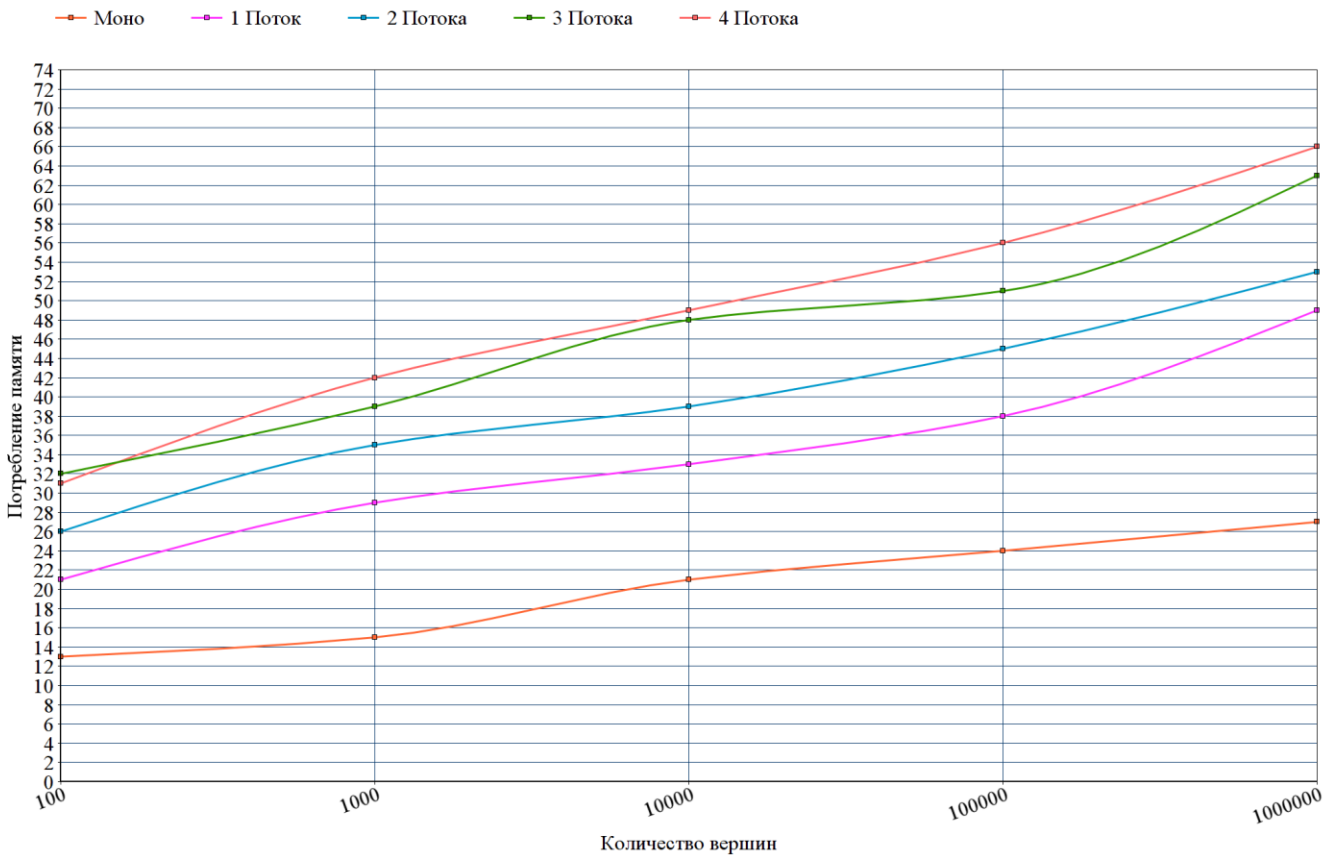

Рис №3.4 «Сравнительная диаграмма эффективности алгоритмов по загруженности процессор

Очевидно, что многопоточные реализации в процессе своей работы требуют большего количества памяти, нежели однопоточное решение. Это объясняется тем, что потоки по своей сути – это совокупность вычислительных ресурсов. Чем больше потоков используется программой, тем больше оперативной памяти необходимо для поддержания потков.

Заключение

В результате работы мною было реализованы и исследованы однопоточная и многопоточная реализация алгоритма поиска кратчайшего пути в графе. Во время работы я повторил основы теории графов, изучил принципы многопоточного программирования, понял его достоинства и недостатки.

Изученный мною алгоритм широко применяется сегодня в различных областях науки и техники, а методы многопоточного программирования в последнее время все чаще используются в больших системах начиная от приложений на мобильных устройствах, заканчивая системами, ответственными за распределения электроэнергии в национальных масштабах.

Список используемой литературы

1. Ресурс [Электронный ресурс]: Имя материала URL: адрес

Приложение

Листинг №1 – Класс, представляющий ребро для однопоточного алгоритма

```
package dijkstra.model;

/**
 * @author Artem Karnov @date 2.05.2017.
 *         artem.karnov@t-systems.com
 *         <p>
 *         Class represents graph's vertex
 */
public class Edge {
    private final Vertex source;
    private final Vertex target;
    private final int weight;

    public Edge(Vertex source, Vertex target, int weight) {
        this.source = source;
        this.target = target;
        this.weight = weight;
    }

    public Vertex getTarget() {
        return target;
    }

    public Vertex getSource() {
        return source;
    }

    public int getWeight() {
        return weight;
    }

    public boolean isConnected(Vertex vertex, Vertex target) {
        return getSource().equals(vertex) && getTarget().equals(target);
    }

    @Override
    public String toString() {
        return source + " " + target;
    }
}
```

Листинг №2 – Класс, представляющий граф для однопоточного алгоритма

```
package dijkstra.model;

import java.util.ArrayList;
import java.util.List;

/**
 * @author Artem Karnov @date 2.05.2017.
 *         artem.karnov@t-systems.com
 *         <p>
 *         Class represents graph and his functional options
 */
public class Graph {
```

```
    private List<Edge> edges;

    public Graph() {
        edges = new ArrayList();
    }

    public List<Edge> getEdges() {
        return edges;
    }

    public void addConnection(Vertex source, Vertex destination, int weight) {
        edges.add(new Edge(source, destination, weight));
    }
}
```

Листинг №3 – Класс, представляющий вершину для однопоточного алгоритма

```
package dijkstra.model;

import java.util.ArrayList;
import java.util.List;

/**
 * @author Artem Karnov @date 2.05.2017.
 *         artem.karnov@t-systems.com
 *         <p>
 *         Class represents graph's edge
 */
public class Vertex {
    final private String name;

    public Vertex(String name) {
        this.name = name;
    }

    public List<Vertex> getNeighbours(List<Edge> edges) {
        List<Vertex> neighbors = new ArrayList();
        for (Edge edge : edges) {
            if (edge.getSource().equals(this)) {
                neighbors.add(edge.getTarget());
            }
        }
        return neighbors;
    }

    @Override
    public String toString() {
        return name;
    }
}
```

Листинг №4 – Класс, интерфейс для реализации однопоточного алгоритма

```
package dijkstra.algorithms;

import dijkstra.model.Vertex;

import java.util.List;
```

```java
/**
 * @author Artem Karnov @date 11.05.2017.
 *         artem.karnov@t-systems.com
 *         <p>
 *         Interface for polimorphic algorithm implementation
 */
public interface Algorithm {
    void execute(Vertex vertexFrom, Vertex vertexTo);

    String getRefactoredPath();

    List<Vertex> getPath();

    int getShortestDistanceValue();
}
```

Листинг №5 – Класс, имплементацию однопоточного алгоритма

```java
package dijkstra.algorithms;


import dijkstra.model.Edge;
import dijkstra.model.Graph;
import dijkstra.model.Vertex;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Set;

import static java.lang.Integer.MAX_VALUE;
import static java.util.Collections.reverse;

/**
 * @author Artem Karnov @date 11.05.2017.
 *         artem.karnov@t-systems.com
 *         <p>
 *         Implementation of Dijkstra's algoritm
 */
public class Dijkstra implements Algorithm {
    private final Graph graph;
    private Set<Vertex> settledVertexes;
    private Map<Vertex, Vertex> predecessors;
    private Map<Vertex, Integer> distanceToVertexes;
    private List<Vertex> resultPath;

    /**
     * Constructor for graph initialization
     *
     * @param graph graph for algorithm executing
     */
    public Dijkstra(Graph graph) {
        this.graph = graph;
    }

    /**
     * Method for running algorithm executing
     *
     * @param vertexFrom source vertex
```

```
     * @param vertexTo   target vertex
     */
    @Override
    public void execute(Vertex vertexFrom, Vertex vertexTo) {
        settledVertexes = new HashSet();
        distanceToVertexes = new HashMap();
        predecessors = new HashMap();

        distanceToVertexes.put(vertexFrom, 0);
        while (!getUnsettledVertexes().isEmpty()) {
            Vertex vertex =
getVertexWithMinimalDistanceToSourceFromUnsettledVertexes();
            settledVertexes.add(vertex);
            findMinimalDistancesToNeighbors(vertex);
        }
        boolean isConnected = predecessors.containsKey(vertexTo);
        if (isConnected) {
            resultPath = constructPath(vertexTo);
        } else {
            resultPath = new ArrayList<>();
        }
    }

    /**
     * Method for getting refactored path
     *
     * @return string which has view Vertex_1 -> Vertex_2 ...
     */
    @Override
    public String getRefactoredPath() {
        if (resultPath == null)
            return "Path isn't real";
        StringBuilder stringBuilder = new StringBuilder();
        for (int i = 0; i < resultPath.size(); i++) {
            stringBuilder.append(resultPath.get(i));
            if (i < resultPath.size() - 1)
                stringBuilder.append("->");
        }
        return stringBuilder.toString();
    }

    /**
     * Method for getting path as list of vertexes
     *
     * @return sorted path of vertexes
     */
    @Override
    public List<Vertex> getPath() {
        return resultPath;
    }

    /**
     * Getting shortest distance
     *
     * @return shortest distance
     */
    @Override
    public int getShortestDistanceValue() {
        int distance = 0;
        if (resultPath == null)
            return 0;
```

```
        for (int i = 0; i < resultPath.size() - 1; i++) {
            distance += getDistance(resultPath.get(i), resultPath.get(i + 1));
        }
        return distance;
    }

    /**
     * Initialization of lists with minimal distances
     *
     * @param vertex vertex for calculation
     */
    private void findMinimalDistancesToNeighbors(Vertex vertex) {
        for (Vertex target : getNeighbors(vertex)) {
            int distanceToTargetViaVertex = getShortestDistance(vertex) +
getDistance(vertex, target);
            if (getShortestDistance(target) > distanceToTargetViaVertex) {
                distanceToVertexes.put(target, distanceToTargetViaVertex);
                predecessors.put(target, vertex);
            }
        }
    }

    /**
     * Getting distance between source and target vertexes
     *
     * @param source source vertex
     * @param target target vertex
     * @return if right path exists - distance between source and target vertexes
     * if right path doesn't exist - Integer.Max value (2147483647)
     */
    private int getDistance(Vertex source, Vertex target) {
        for (Edge edge : getEdges()) {
            if (edge.isConnected(source, target)) {
                return edge.getWeight();
            }
        }
        return MAX_VALUE;
    }

    /**
     * Getting nearest neighbours of vertex
     *
     * @param vertex vertex for getting neighbours
     * @return list of vertex's nearest neighbours
     */
    private List<Vertex> getNeighbors(Vertex vertex) {
        List<Vertex> neighbors = vertex.getNeighbours(getEdges());
        neighbors.removeAll(settledVertexes);
        return neighbors;
    }

    /**
     * Getting vertex with minimal distention to source vertex from unsettled
vertexes
     *
     * @return vertex with minimal distention to source vertex from unsettled
vertexes
     */
    private Vertex getVertexWithMinimalDistanceToSourceFromUnsettledVertexes() {
        Vertex minimum = null;
```

```
        for (Vertex vertex : getUnsettledVertexes()) {
            if (getShortestDistance(vertex) < getShortestDistance(minimum)) {
                minimum = vertex;
            }
        }
        return minimum;
    }

    /**
     * Getting distance to adjusted vertex
     *
     * @param destination target vertex
     * @return if right path exists - shortest distance to target vertexes
     * if right path doesn't exist - Integer.Max value (2147483647)
     */
    private int getShortestDistance(Vertex destination) {
        return distanceToVertexes.containsKey(destination) ?
                distanceToVertexes.get(destination) : MAX_VALUE;
    }

    /**
     * Constructing path list according to predecessors list
     *
     * @param step target vertex
     * @return shortest path to target vertex
     */
    private List<Vertex> constructPath(Vertex step) {
        List<Vertex> path = new LinkedList();
        path.add(step);
        while (predecessors.containsKey(step)) {
            step = predecessors.get(step);
            path.add(step);
        }
        reverse(path);
        return path;
    }

    /**
     * Getting list of all edges of graph
     *
     * @return list of all edges of graph
     */
    private List<Edge> getEdges() {
        return graph.getEdges();
    }

    /**
     * Getting set of unsettled vertexes
     *
     * @return set of unsettled vertexes
     */
    private Set<Vertex> getUnsettledVertexes() {
        Set<Vertex> vertices = new HashSet(distanceToVertexes.keySet());
        vertices.removeAll(settledVertexes);
        return vertices;
    }
}
```

Листинг №6 – Класс, представляющий возможность создания вершины для теста

```
package dijkstra.utility;

import dijkstra.model.Vertex;

/**
 * @author Artem Karnov @date 11.05.2017.
 *         artem.karnov@t-systems.com
 */

public class VertexUtility {
    public static Vertex createVertex(int number) {
        return new Vertex("Vertex_" + number);
    }
}
```

Листинг №7 – Тестирование корректности работы однопоточного алгоритма

```
package dijkstra.test;

import dijkstra.algorithms.Algorithm;
import dijkstra.algorithms.Dijkstra;
import dijkstra.model.Graph;
import dijkstra.model.Vertex;
import dijkstra.utility.VertexUtility;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

/**
 * @author Artem Karnov @date 2.05.2017.
 *         artem.karnov@t-systems.com
 *         <p>
 *         Class represents for correction unit testing
 */
public class AlgorithmCorrectionTest {
    private Graph graph;
    private List<Vertex> vertices;

    @Before
    public void setUp() {
        graph = new Graph();
        vertices = new ArrayList<>();

        vertices = Arrays.asList(
                VertexUtility.createVertex(0),
                VertexUtility.createVertex(1),
                VertexUtility.createVertex(2),
                VertexUtility.createVertex(3),
                VertexUtility.createVertex(4),
                VertexUtility.createVertex(5)
        );

        graph.addConnection(vertices.get(1), vertices.get(2), 10);
```

```
        graph.addConnection(vertices.get(1), vertices.get(3), 20);
        graph.addConnection(vertices.get(2), vertices.get(4), 50);
        graph.addConnection(vertices.get(2), vertices.get(5), 6);
        graph.addConnection(vertices.get(5), vertices.get(4), 6);
    }

    @Test
    public void algorithmCorrectionTest() {
        Algorithm dijkstra = new Dijkstra(graph);
        dijkstra.execute(vertices.get(1), vertices.get(4));

        List<Vertex> actualResult = dijkstra.getPath();
        List<Vertex> expectedResult = Arrays.asList(
                vertices.get(1),
                vertices.get(2),
                vertices.get(5),
                vertices.get(4)
        );

        Assert.assertEquals(expectedResult, actualResult);
        Assert.assertEquals(22, dijkstra.getShortestDistanceValue());
    }

    @Test
    public void algorithmCorrectionCornerTest() {
        Algorithm dijkstra = new Dijkstra(graph);
        dijkstra.execute(vertices.get(3), vertices.get(5));

        List<Vertex> actualResult = dijkstra.getPath();
        List<Vertex> expectedResult = Arrays.asList();

        Assert.assertEquals(expectedResult, actualResult);
        Assert.assertEquals(0, dijkstra.getShortestDistanceValue());
    }

}
```

Листинг №8 – Тестирование производительности однопоточного алгоритма

```
package dijkstra.test;



import dijkstra.algorithms.Algorithm;
import dijkstra.algorithms.Dijkstra;
import dijkstra.model.Graph;
import dijkstra.model.Vertex;
import org.junit.Before;
import org.junit.Test;

import java.util.ArrayList;
import java.util.List;

/**
 * @author Artem Karnov @date 03.05.2017.
 *         artem.karnov@t-systems.com
 */

public class PerformanceTest {
    private Graph graph;
    private List<Vertex> vertices;
```

```
    private int numberOfVertexes = 100000;

    @Before
    public void setUp() {
        graph = new Graph();
        vertices = new ArrayList<>();

        for (int i = 0; i < numberOfVertexes; i++) {
            vertices.add(new Vertex("Vertex_" + i));
        }

        for (int i = 0; i < numberOfVertexes - 1; i++) {
            graph.addConnection(vertices.get(i), vertices.get(i + 1), (int)
Math.random() * numberOfVertexes);
        }

    }

    @Test
    public void algorithmCorrectionTest() {
        long start = System.currentTimeMillis();
        Algorithm dijkstra = new Dijkstra(graph);
        dijkstra.execute(vertices.get(0), vertices.get(numberOfVertexes - 1));
        System.out.println(dijkstra.getRefactoredPath());
        long finish = System.currentTimeMillis();
        System.out.println(finish - start);
    }
}
```

Листинг №9 – Класс, представляющий ребро для многопоточного алгоритма

```
package dijkstra.model;

/**
 * @author Artem Karnov @date 21.04.2017.
 *         artem.karnov@t-systems.com
 *         <p>
 *         POJO represents graph's edge
 */
public class Edge {
    private final String id;
    private final Vertex source;
    private final Vertex destination;
    private final int weight;

    public Edge(String id, Vertex source, Vertex destination, int weight) {
        this.id = id;
        this.source = source;
        this.destination = destination;
        this.weight = weight;
    }

    public Vertex getDestination() {
        return destination;
    }

    public Vertex getSource() {
        return source;
    }

    public int getWeight() {
```

```
        return weight;
    }

    @Override
    public String toString() {
        return source + " " + destination + " " + weight;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((id == null) ? 0 : id.hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Edge other = (Edge) obj;
        if (id == null) {
            if (other.id != null)
                return false;
        } else if (!id.equals(other.id))
            return false;
        return true;
    }
}
```

Листинг №10 – Класс, представляющий граф для многопоточного алгоритма

```
package dijkstra.model;

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;

/**
 * @author Artem Karnov @date 2.05.2017.
 *         artem.karnov@t-systems.com
 *         <p>
 *         Class represents graph and his functional options
 */
public class Graph {

    private final List<Vertex> vertexes;
    private final List<Edge> edges;
    private Map<Vertex, List<Edge>> connections;

    public Graph(List<Vertex> vertexes, List<Edge> edges) {
        this.vertexes = vertexes;
        this.edges = edges;
    }
```

```
    public static List<Map<Vertex, List<Edge>>> splitConnections(Map<Vertex,
List<Edge>> connections) {

        List<Map<Vertex, List<Edge>>> splittedConnections = new ArrayList();

        Map<Vertex, List<Edge>> firstConnectionsGroup = new HashMap();
        Map<Vertex, List<Edge>> secondConnectionsGroup = new HashMap();

        for (Vertex v : connections.keySet()) {
            List<Edge> edges = new ArrayList(connections.get(v));

            Collections.shuffle(edges);

            int size = edges.size();

            List<Edge> list1 = new ArrayList(edges.subList(0, size / 2));
            firstConnectionsGroup.put(v, list1);

            List<Edge> list2 = new ArrayList(edges.subList(size / 2, size));
            secondConnectionsGroup.put(v, list2);
        }

        splittedConnections.add(Collections.unmodifiableMap(firstConnectionsGroup));
        splittedConnections.add(Collections.unmodifiableMap(secondConnectionsGroup));

        return Collections.unmodifiableList(splittedConnections);
    }

    public List<Vertex> getVertexes() {
        return Collections.unmodifiableList(vertexes);
    }

    public List<Edge> getEdges() {
        return Collections.unmodifiableList(edges);
    }

    private List<Edge> getVertexConnections(final Vertex vertex) {
        List<Edge> vertexConnections = new LinkedList<Edge>();

        for (Edge edge : edges)
            if (edge.getSource().equals(vertex))
                vertexConnections.add(edge);

        return Collections.unmodifiableList(vertexConnections);
    }

    public Map<Vertex, List<Edge>> getConnections() {

        if (connections == null) {
            connections = new HashMap();
            for (Vertex vertex : vertexes)
                connections.put(vertex, getVertexConnections(vertex));

        }

        return Collections.unmodifiableMap(connections);
    }

}
```

Листинг №11 – Класс, представляющий вершину для многопоточного алгоритма

```
package dijkstra.model;

/**
 * @author Artem Karnov @date 2.05.2017.
 *         artem.karnov@t-systems.com
 *         <p>
 *         POJO represents graph's vertex
 */
public class Vertex {

    final private String id;

    public Vertex(String id) {
        this.id = id;
    }

    public String getId() {
        return id;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((id == null) ? 0 : id.hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Vertex other = (Vertex) obj;
        if (id == null) {
            if (other.id != null)
                return false;
        } else if (!id.equals(other.id))
            return false;
        return true;
    }

    @Override
    public String toString() {
        return id;
    }
}
```

Листинг №12 – Класс, представляющий не посещённую вершину для многопоточного алгоритма

```
package dijkstra.model;

/**
 * @author Artem Karnov @date 15.05.2017.
```

```
 *         artem.karnov@t-systems.com
 *         <p>
 *         Class represdents unseteled vertex
 */
public class UnsettledVertex implements Comparable<UnsettledVertex> {
    private int distance;
    private Vertex vertex;
    private Vertex predecessor;

    public UnsettledVertex(final int distance, final Vertex vertex, final Vertex
predecessor) {
        this.distance = distance;
        this.vertex = vertex;
        this.predecessor = predecessor;
    }

    public int getDistance() {
        return distance;
    }

    public Vertex getVertex() {
        return vertex;
    }

    public Vertex getPredecessor() {
        return predecessor;
    }

    @Override
    public int compareTo(UnsettledVertex other) {
        return (distance - other.distance);
    }

    public boolean isFartherAwayThan(final UnsettledVertex other) {
        return (this.compareTo(other) > 0);
    }

}
```

Листинг №13 – Класс представляющий, интерфейс для реализации для многопоточного алгоритма

```
package dijkstra.algorithms;

import dijkstra.model.Vertex;

import java.util.List;

/**
 * @author Artem Karnov @date 2.05.2017.
 *         artem.karnov@t-systems.com
 *         <p>
 *         Interface for classes represent algorithm's implementations
 */
public interface Algorithm {
    void execute(Vertex vertexFrom, Vertex vertexTo);

    String getRefactoredPath();

    List<Vertex> getPath();
}
```

Листинг №14 –Имплементация многопоточного алгоритма

```
package dijkstra.algorithms;

import dijkstra.model.Edge;
import dijkstra.model.Graph;
import dijkstra.model.UnsettledVertex;
import dijkstra.model.Vertex;

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.PriorityQueue;
import java.util.Queue;
import java.util.Set;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.RecursiveAction;

/**
 * @author Artem Karnov @date 2.05.2017.
 *         artem.karnov@t-systems.com
 *         <p>
 *         Asynchron implementation of Dijkstra's algorithm
 */
public class AsynchronousDijkstra implements Algorithm {
    private static final int MAX_PROCESSING_SPLIT_COUNT = 2;
    private static ForkJoinPool joinPool =
dijkstra.utility.Concurrency.getForkJoinPool();
    private final List<Vertex> vertexes;
    private List<Vertex> resultPath;
    private Set<Vertex> settledVertexes;
    private Map<Vertex, Vertex> predecessors;
    private ProcessingTask rootProcessingTask;
    private List<UnsettledVertex> newUnsettledVertexes;
    private UnsettledVertex winner;
    private List<ProcessingTask> processingTasks;
    private CyclicBarrier processingTaskBarrier;
    private CyclicBarrier reexecutingTaskBarrier;
    private CyclicBarrier leavesDoneBarrier;
    private BlockingQueue<Integer> notifyQueue;
    private int leafProcessingTaskCount;

    /**
     * Constructor for initialization data structures
     *
     * @param graph graph for calculation
     */
    public AsynchronousDijkstra(final Graph graph) {
        vertexes = graph.getVertexes();
        settledVertexes = new HashSet();
        predecessors = new HashMap();
```

```
        newUnsettledVertexes = new ArrayList();
        processingTasks = new ArrayList();
        rootProcessingTask = new ProcessingTask(graph.getConnections());
        notifyQueue = new LinkedBlockingQueue();
    }

    /**
     * Method for algorithm executing
     *
     * @param vertexFrom begin vertex
     * @param vertexTo   finish vertex
     */
    @Override
    public void execute(Vertex vertexFrom, Vertex vertexTo) {
        settledVertexes.clear();
        predecessors.clear();

        if (null != rootProcessingTask) {
            rootProcessingTask.setSource(vertexFrom);
            if (rootProcessingTask.isActive == false)
                joinPool.execute(rootProcessingTask);
            else
                rootProcessingTask.compute();
            try {
                notifyQueue.take();
            } catch (Exception e) {
                return;
            }
        }
        LinkedList<Vertex> path = new LinkedList();
        Vertex step = vertexTo;
        if (predecessors.get(step) == null)
            return;
        path.add(step);
        while (predecessors.get(step) != null) {
            step = predecessors.get(step);
            path.add(step);
        }
        Collections.reverse(path);
        resultPath = Collections.unmodifiableList(path);
    }

    /**
     * Method for path refactoring
     *
     * @return refactored path
     */
    @Override
    public String getRefactoredPath() {
        if (resultPath == null)
            return "path isn't real";
        StringBuilder stringBuilder = new StringBuilder();
        for (int i = 0; i < resultPath.size(); i++) {
            stringBuilder.append(resultPath.get(i));
            if (i < resultPath.size() - 1)
                stringBuilder.append("->");
        }
        return stringBuilder.toString();
    }

    /**
```

```
     * Getting path which consists of vertexes
     *
     * @return list of vertexes
     */
    @Override
    public List<Vertex> getPath() {
        return resultPath;
    }

    /**
     * Inner class represent task for processing
     */
    private class ProcessingTask extends RecursiveAction {
        private ProcessingTask firstProcessingTask;
        private ProcessingTask secondProcessingTask;
        private Map<Vertex, List<Edge>> connections;
        private Map<Vertex, Integer> distancesFromSource;
        private Queue<UnsettledVertex> unsettledNodesQueue;
        private List<UnsettledVertex> newUnsettledVertices;
        private boolean leafTask = true;
        private boolean isActive;
        private Runnable queryTasksForWinner = new Runnable() {
            public void run() {
                queryTasksForWinner();
            }
        };
        private Runnable signalLeavesDone = new Runnable() {
            public void run() {
                try {
                    notifyQueue.put(1);
                } catch (Exception e) {
                    return;
                }
            }
        };

        public ProcessingTask(final Map<Vertex, List<Edge>> connections) {
            this(connections, 0);
        }

        /**
         * Task processing with known data
         *
         * @param connections vertexes connections
         * @param level       level of task
         */
        public ProcessingTask(final Map<Vertex, List<Edge>> connections, final int
level) {
            if (level < MAX_PROCESSING_SPLIT_COUNT) {
                List<Map<Vertex, List<Edge>>> splitConnections =
Graph.splitConnections(connections);
                firstProcessingTask = new ProcessingTask(splitConnections.get(0),
level + 1);
                secondProcessingTask = new ProcessingTask(splitConnections.get(1),
level + 1);
                leafTask = false;
            } else {
                this.connections = connections;
            }

            distancesFromSource = new HashMap();
```

```
            unsettledNodesQueue = new PriorityQueue();
            newUnsettledVertices = new ArrayList();

            if (leafTask) {
                leafProcessingTaskCount++;
                processingTaskBarrier = new CyclicBarrier(leafProcessingTaskCount,
queryTasksForWinner);
                leavesDoneBarrier = new CyclicBarrier(leafProcessingTaskCount,
signalLeavesDone);
                reexecutingTaskBarrier = new CyclicBarrier(leafProcessingTaskCount +
1);
            }
            processingTasks.add(this);
        }

        public void setSource(final Vertex source) {
            UnsettledVertex unsettledVertex = new UnsettledVertex(0, source, null);
            winner = unsettledVertex;
            settledVertexes.add(source);
        }

        private void findWinner() {
            try {
                processingTaskBarrier.await();
            } catch (InterruptedException ex) {
                return;
            } catch (BrokenBarrierException ex) {
                return;
            }
            processWinnerAndUnsettledNodes();
        }

        private void queryTasksForWinner() {
            UnsettledVertex potentialWinner = null;
            newUnsettledVertexes.clear();

            for (ProcessingTask pt : processingTasks) {
                UnsettledVertex currentUnsettledVertex =
pt.unsettledNodesQueue.peek();

                if (null == currentUnsettledVertex)
                    continue;

                if ((null == potentialWinner) ||
potentialWinner.isFartherAwayThan(currentUnsettledVertex))
                    potentialWinner = currentUnsettledVertex;

                newUnsettledVertexes.addAll(pt.newUnsettledVertices);
            }

            winner = potentialWinner;

            if (null != winner) {
                settledVertexes.add(winner.getVertex());
                predecessors.put(winner.getVertex(), winner.getPredecessor());
            }
        }

        private void processWinnerAndUnsettledNodes() {
            if ((null != winner) && (0 == winner.getDistance()))
                reset(winner.getVertex());
```

```
            else if (winner == unsettledNodesQueue.peek())
                unsettledNodesQueue.poll();

            for (UnsettledVertex currentVertex : newUnsettledVertexes)
                distancesFromSource.put(currentVertex.getVertex(),
currentVertex.getDistance());
        }

        private void reset(final Vertex vertex) {
            unsettledNodesQueue.clear();
            distancesFromSource.clear();
            distancesFromSource.put(vertex, 0);
        }

        @Override
        public void compute() {
            isActive = true;

            boolean reexecuting = !inForkJoinPool();

            while (true) {
                if (false == reexecuting) {
                    if (leafTask) {
                        processWinnerAndUnsettledNodes();
                        while (null != winner) {
                            relax(winner.getVertex(), winner.getDistance());
                            findWinner();
                        }

                    } else {
                        firstProcessingTask.fork();
                        secondProcessingTask.compute();
                        break;
                    }

                    try {
                        leavesDoneBarrier.await();
                    } catch (InterruptedException ex) {
                        return;
                    } catch (BrokenBarrierException ex) {
                        return;
                    }
                }

                try {
                    reexecutingTaskBarrier.await();
                } catch (InterruptedException ex) {
                    return;
                } catch (BrokenBarrierException ex) {
                    return;
                }

                if (reexecuting) break;
            }
        }

        private void relax(final Vertex node, int distToNode) {
            newUnsettledVertices.clear();

            for (Vertex target : getNeighbors(node)) {
                int dist = distToNode + getDistance(node, target);
```

```
                if (getShortestDistance(target) > dist) {
                    distancesFromSource.put(target, dist);
                    newUnsettledVertices.add(new
UnsettledVertex(getShortestDistance(target) + distToNode, target, node));
                }
            }
            unsettledNodesQueue.addAll(newUnsettledVertices);
        }

        private int getShortestDistance(final Vertex destination) {
            Integer shortestDistance = distancesFromSource.get(destination);
            return (shortestDistance == null) ? Integer.MAX_VALUE : shortestDistance;
        }

        private int getDistance(final Vertex node, final Vertex target) {
            for (Edge edge : connections.get(node))
                if (edge.getDestination().equals(target))
                    return edge.getWeight();
            return 0;
        }

        private List<Vertex> getNeighbors(final Vertex node) {
            List<Vertex> nodeNeighbors = new ArrayList();
            for (Edge edge : connections.get(node)) {
                Vertex destination = edge.getDestination();
                if (!settledVertexes.contains(destination))
                    nodeNeighbors.add(destination);
            }
            return Collections.unmodifiableList(nodeNeighbors);
        }
    }
}
```

Листинг №15 – Класс, представляющий возможность создание потоков

```
package dijkstra.utility;

import java.util.concurrent.ForkJoinPool;

/**
 * @author Artem Karnov @date 2.05.2017.
 *         artem.karnov@t-systems.com
 *         <p>
 *         Utility class for thread pool cointainer
 */
public class Concurrency {

    static ForkJoinPool fork_join_pool = new ForkJoinPool();

    public static ForkJoinPool getForkJoinPool() {
        return fork_join_pool;
    }
}
```

Листинг №16 – Тестирование корректности работы многопоточного алгоритма

```
package dijkstra.test;

import dijkstra.algorithms.Algorithm;
```

```java
import dijkstra.algorithms.AsynchronousDijkstra;
import dijkstra.model.Edge;
import dijkstra.model.Graph;
import dijkstra.model.Vertex;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

import java.util.ArrayList;
import java.util.List;


public class AlgorithmCorrectionTest {
    private List<Vertex> nodes;
    private List<Edge> edges;
    private Graph graph;

    @Before
    public void setUp() {
        nodes = new ArrayList();
        edges = new ArrayList();

        nodes.add(new Vertex("0"));
        nodes.add(new Vertex("1"));
        nodes.add(new Vertex("2"));
        nodes.add(new Vertex("3"));
        nodes.add(new Vertex("4"));
        nodes.add(new Vertex("5"));
        nodes.add(new Vertex("6"));
        nodes.add(new Vertex("7"));
        nodes.add(new Vertex("8"));
        nodes.add(new Vertex("9"));
        nodes.add(new Vertex("10"));

        edges.add(new Edge("1-2", nodes.get(1), nodes.get(2), 10));
        edges.add(new Edge("1-3", nodes.get(1), nodes.get(3), 20));
        edges.add(new Edge("1-4", nodes.get(1), nodes.get(4), 50));
        edges.add(new Edge("5-4", nodes.get(5), nodes.get(4), 6));
        edges.add(new Edge("2-5", nodes.get(2), nodes.get(5), 4));
        graph = new Graph(nodes, edges);
    }


    @Test
    public void SimpleAsynchronousAlgorithmCorrectionTest() {
        Algorithm dijkstra = new AsynchronousDijkstra(graph);
        dijkstra.execute(nodes.get(1), nodes.get(4));
        Assert.assertEquals("1->2->5->4", dijkstra.getRefactoredPath());
    }


    @Test
    public void SimpleAsynchronousAlgorithmEdgeTest() {
        Algorithm dijkstra = new AsynchronousDijkstra(graph);
        dijkstra.execute(nodes.get(1), nodes.get(6));
        Assert.assertEquals("path isn't real", dijkstra.getRefactoredPath());
        ;
    }


}
```

Листинг №17 – Тестирование производительности многопоточного алгоритма

```
package dijkstra.test;

import dijkstra.algorithms.Algorithm;
import dijkstra.algorithms.AsynchronousDijkstra;
import dijkstra.model.Edge;
import dijkstra.model.Graph;
import dijkstra.model.Vertex;
import org.junit.Before;
import org.junit.Test;

import java.util.ArrayList;
import java.util.List;

/**
 * @author Artem Karnov @date 15.05.2017.
 *         artem.karnov@t-systems.com
 */

public class PerformanceTest {
    private List<Vertex> vertices;
    private List<Edge> edges;
    private Graph graph;
    private int numberOfVertexes = 100;

    @Before
    public void setUp() {
        vertices = new ArrayList();
        edges = new ArrayList();

        for (int i = 0; i < numberOfVertexes; i++) {
            vertices.add(new Vertex("Vertex_" + i));
        }

        for (int i = 0; i < numberOfVertexes - 1; i++) {
            edges.add(new Edge(i + " - " + i + 1, vertices.get(i), vertices.get(i +
1), (int) Math.random() * numberOfVertexes));
        }
        graph = new Graph(vertices, edges);

    }

    @Test
    public void SimpleAsynchronousAlgorithmCorrectionTest() {
        long start = System.currentTimeMillis();
        Algorithm dijkstra = new AsynchronousDijkstra(graph);
        dijkstra.execute(vertices.get(0), vertices.get(numberOfVertexes - 1));
        System.out.println(dijkstra.getRefactoredPath());
        long finish = System.currentTimeMillis();
        System.out.println(finish - start);
    }
}
```