УДК 519.178
ББК 22.176

АЛГОРИТМ РЕШЕНИЯ ДИНАМИЧЕСКОЙ ЗАДАЧИ ПОИСКА КРАТЧАЙШИХ РАССТОЯНИЙ В ГРАФЕ

Ураков А. Р.[1], **Тимеряев Т. В.**[2]
(*Уфимский государственный авиационный технический университет, Уфа*)

Рассматривается полностью динамическая задача поиска кратчайших расстояний между всеми парами вершин неориентированного графа. Для данной задачи предлагается метод решения, учитывающий все возможные изменения расстояний в графе с помощью процедур добавления и удаления ребра. Для процедуры удаления ребра предложен алгоритм, использующий понятие точек, равноудаленных от вершин, инцидентных удаляемому ребру. Алгоритм позволяет существенно уменьшить время решения и объем используемой памяти в практических сценариях. Для доказательства практической эффективности предложенного метода решения произведен вычислительный эксперимент с использованием известных быстрых статических и динамических алгоритмов.

Ключевые слова: динамические кратчайшие расстояния, актуализация графа, коррекция графа, равноудаленные точки.

Введение

Задачи, связанные с поиском кратчайших путей (и расстояний, ассоциированных с этими путями), являются одними из самых исследованных в теории графов. В классической (*статической*) задаче поиска кратчайших путей между всеми вершинами графа (задача APSP) предполагается, что исследуемый граф – его

[1] *Айрат Ренатович Ураков, кандидат физико-математических наук, доцент* (*urakov@ufanet.ru*).
[2] *Тимофей Валерьевич Тимеряев* (*timeryaev@yandex.ru*).

структура и веса ребер (дуг) – не меняются с течением времени. В этом случае кратчайшие пути находятся в графе один раз и в дальнейшем используются без изменений. Для статической задачи разработано большое число изящных алгоритмов с различными областями эффективного применения, среди самых известных из которых алгоритмы Беллмана – Форда, Флойда – Уоршелла, Дейкстры и др. В качестве меры эффективности алгоритма, как правило, используют скорость нахождения решения.

Другой, менее исследованной является *динамическая* разновидность задачи. В динамической задаче предполагается, что подлежащий граф – его структура и веса ребер (дуг) – изменяются с течением времени, в связи с чем необходимо поддерживать вычисленные кратчайшие пути в актуальном состоянии с учетом происходящих изменений. Различают две разновидности динамической задачи: *полностью динамическая* – допустимо увеличение и уменьшение весов ребер, *частично динамическая* – допустимо либо увеличение, либо уменьшение весов ребер. Здесь, как и в случае со статической задачей, важна скорость решения, поэтому эффективное применение методов решения статической задачи – расчет кратчайших путей заново при изменении графа – сильно ограничено (размерностью графа, частотой изменений графа).

В статье предлагается метод решения полностью динамической задачи поиска кратчайших расстояний между всеми вершинами ненаправленного графа с положительными вещественными весами ребер. Следует подчеркнуть, что при решении данной задачи нас интересует лишь актуальность кратчайших расстояний между вершинами, информация о соответствующих путях не актуализируется. Метод решения основан на использовании двух алгоритмов: удаления и добавления ребра, с помощью которых учитываются все возможные модификации графа. Практическая ценность изложенных в статье метода и алгоритмов заключается в том, что в сравнении с известными быстрыми алгоритмами они позволяют решать задачу актуализации расстояний после удаления ребра на разреженных графах быстрее и при этом не используют никакой дополнительной информации кроме рассто-

яний, что позволяет существенно уменьшить использование памяти при решении.

1. Термины и обозначения

В статье используются следующие термины и обозначения.

$G(V, E, w)$ – неориентированный граф G с множеством вершин V, множеством ребер E и весовой функцией на ребрах $w : E \mapsto \mathbb{R}_{>0}$.

$e(a_i, b_j)$ – ребро между вершинами a_i и b_j.

$w(a_i, b_j)$ – вес ребра между вершинами a_i и b_j. Так как граф неориентированный, то $w(a_i, b_j) = w(b_j, a_i)$.

Расстояние между a_i и b_j – кратчайшее расстояние между вершинами a_i и b_j.

$l(a_i, b_j)$ – расстояние между вершинами a_i и b_j, так как граф неориентированный $l(a_i, b_j) = l(b_j, a_i)$.

$[a_i, \dots, b_j]$ – путь от вершины a_i до вершины b_j.

Свойство оптимальности кратчайшего пути – каждый подпуть кратчайшего пути является также кратчайшим, т.е. если $[a_i, \dots, b_j]$ кратчайший путь между a_i и b_j, то любой подпуть $[a_x, \dots, b_y]$ этого пути будет кратчайшем путем между a_x и b_y.

2. Существующие алгоритмы

Первые алгоритмы решения динамических задач поиска кратчайших путей опубликованы в 60–70-х годах прошлого столетия [1, 8, 18, 19]. Алгоритмы в этих работах на основе свойств кратчайших путей определяют пары вершин, между которыми в результате изменения весов ребер могли измениться расстояния. Расстояния между найденными парами пересчитывается с использованием простых алгебраических и логических операций, а также статических алгоритмов поиска кратчайших путей. Временные сложности этих алгоритмов в худшем случае определяются случаем увеличения веса ребра и не дают улучшения по сравнению с полным пересчетом расстояний графа с использованием статических методов.

Авторы [22] представили полностью динамический алгоритм, поддерживающий множественные изменения весов ребер между актуализациями расстояний. В [12] предложили полностью динамический алгоритм для задачи с единственным источником и наибольшей эффективностью на планарных графах и графах с ограниченной степенью вершин. Алгоритмы, улучшающие временную сложность полного пересчета расстояний, были предложены для графов с ограниченными весами ребер [3, 17] и планарных графов [11, 16]. В [7] авторы переложили полностью динамический алгоритм для направленных графов с неотрицательными весами с амортизированным временем $O(n^2 log^3|V|)$ для любой последовательности модификаций графа.

Алгоритмы приближенного решения задачи рассмотрены в [4, 15, 24]. Многие алгоритмы, решающие динамическую задачу, хранят вспомогательную информацию, используемую при пересчете расстояний, что требует большого объема оперативной памяти при практической реализации и тем самым ограничивает размерность используемых графов. Для возможности работы с большими графами рядом авторов предложены алгоритмы [14, 21], хранящие вспомогательную информацию под управлением СУБД. Распределенные алгоритмы решения задачи рассматриваются в [5, 23].

3. Метод коррекции графа

Изменение взвешенного графа состоит из операций следующего вида: изменение множества вершин, изменение множества ребер и изменение весов ребер. Требуемое изменение графа легко разложить на последовательность простейших операций, т.е. таких, которые затрагивают только одну вершину или ребро. Если мы будем отслеживать изменения расстояний при каждой простейшей операции, мы будем знать результирующие расстояния рассматриваемого графа.

Добавление/удаление вершины a, несвязанной с другими, изменяет расстояния в графе тривиальным образом – добавляются/удаляются равные бесконечности расстояния между a и всеми

остальными вершинами графа. Операцию изменения списка вершин, связанных с другими вершинами графа, можно свести к операции изменения списка ребер.

Заметим, что если пара вершин a и b связана ребром веса w_1, то удаление или добавление ребра, связывающего a и b веса w_2, не повлияет на расстояния, если $w_2 > w_1$. Отсюда следует, что уменьшение веса ребра можно свести к процедуре добавления ребра, а увеличение веса ребра можно свести к процедуре удаления ребра. В первом случае сначала добавляется новое ребро меньшего веса, далее удаляется существующее ребро большего веса, что не повлияет на расстояния (и не требует пересчета). Во втором случае сперва добавляется новое ребро большего веса, что не влияет на расстояния (и не требует пересчета), после этого удаляется существующее ребро меньшего веса.

Таким образом, рассматриваемая задача актуализации расстояний в динамическом графе может быть решена с помощью процедур удаления и добавления ребра. Далее дается описание алгоритмов этих процедур.

4. Добавление ребра

Рассматривается добавление ребра $e(a_0, b_0)$ веса $w(a_0, b_0)$ между вершинами a_0 и b_0. Необходимо пересчитать расстояния $l(a_i, b_j)$, которые в результате добавления данного ребра уменьшатся. Предполагается, что вес добавляемого ребра меньше текущего расстояния между вершинами $w(a_0, b_0) < l(a_0, b_0)$, так как в противном случае пересчет расстояний производить не нужно.

Добавляемое ребро $e(a_0, b_0)$ можно представлять как «мост», соединяющий множества вершин $A = \{a_i\}$ и $B = \{b_j\}$, которые разбивают множество вершин графа V на вершины, которые ближе к a_0, и вершины, которые ближе к b_0:

$$\begin{aligned} A &= \{v \in V | l(v, a_0) \leqslant l(v, b_0)\}, \\ B &= \{v \in V | l(v, b_0) \leqslant l(v, a_0)\}. \end{aligned} \tag{1}$$

В случае равенства расстояний до a_0 и до b_0 вершина v может быть помещена в любое из множеств A, B. Будем считать, что $a_0 \in A$ и $b_0 \in B$. При разбиении вершин графа (1), добавляя

ребро $e(a_0, b_0)$, достаточно пересчитать расстояния лишь между парами вершин одна из которых принадлежит A, а другая B, т.е. $(v, u) : v \in A, u \in B$. Если обе вершины принадлежат одному из множеств, например, $v, u \in A$, то расстояние между ними не изменится. Это обусловлено двумя факторами. Во-первых, добавление ребра $e(a_0, b_0)$ не изменит принадлежность вершин множеству A: расстояние от них до a_0 все так же будет не больше чем до b_0. Действительно, если бы расстояние от, например, v до b_0 изменилось, соответствующий путь имел бы вид $[v, \ldots, a_0, b_0]$, что означало бы, что $l(v, a_0) < l(v, b_0)$. Во-вторых, кратчайший путь между v и u не может содержать добавляемое ребро $e(a_0, b_0)$, так как расстояние от обоих v и u до a_0 не больше чем до b_0, и кратчайший путь $[v, \ldots, u]$ между v и u в противном случае содержал бы подпуть вида $[v, \ldots, b_0, a_0]$, который не является кратчайшим, что противоречило бы свойству оптимальности кратчайшего пути.

Вершины графа можно поделить между множествами A и B так, что элементы этих множеств вместе со связывающими их ребрами в графе будут образовывать два связных подграфа. Действительно, это можно сделать, например, отнеся все вершины, которые имеют одинаковое расстояние до a_0 и b_0, либо к A, либо к B. Более того, в этом случае можно представить объединение вершин из множеств A и B в виде двух деревьев t_A и t_B с корнями a_0 и b_0 соответственно. Для каждого из множеств A и B (при наличии достаточного числа вершин в них) существует несколько вариантов возможных деревьев с корнями a_0 и b_0. Среди этих вариантов существуют один обладающий полезным свойством.

В *дереве кратчайших путей* с корнем – некоторой вершиной v – длина пути от v до любой другой вершины дерева u равна расстоянию между этими вершинами $l(v, u)$. Если выбрать в качестве деревьев t_A и t_B на множествах A и B деревья кратчайших путей с корнями a_0 и b_0, соответственно, то, используя свойство этих деревьев, можно сократить число пар вершин (a_i, b_j), между которыми необходимо пересчитать расстояния.

Предположим, что вершина a_i дерева t_A в направлении от

корня дерева связана ребрами с вершинам $a_{i+1}, a_{i+2}, \ldots, a_{i+k}$. Также предположим, что в данный момент времени мы пересчитываем расстояние между a_i и некоторой вершиной b_j дерева t_B. Расстояние между a_i и b_j необходимо пересчитать, если длина пути через добавляемое ребро $e(a_0, b_0)$ будет меньше текущего расстояния между a_i и b_j, т.е. если выполняется условие

$$l(a_i, a_0) + w(a_0, b_0) + l(b_0, b_j) < l(a_i, b_j). \tag{2}$$

Если неравенство (2) не выполняется, величина $l(a_i, b_j)$ не пересчитывается, так как существует путь между a_i и b_j, не проходящий через $e(a_0, b_0)$ и обладающий не большей длиной. Но в этом случае не надо также пересчитывать расстояния между b_j и вершинами $a_{i+1}, a_{i+2}, \ldots, a_{i+k}$.

Действительно, так как дерево t_A является деревом кратчайших путей, то в случае пересчета расстояния между, например, a_{i+1} и b_j соответствующий кратчайший путь имел бы вид

$$[a_{i+1}, a_i, \ldots, a_0, b_0, \ldots, b_j]. \tag{3}$$

Если расстояние между a_i и b_j не было пересчитано, потому что выражение (2) имело форму равенства, то нет необходимости пересчитывать и расстояние между a_{i+1} и b_j, так как оно не изменится (не изменится длина подпути $[a_i, \ldots, b_j]$). Если же путь между a_i и b_j через $e(a_0, b_0)$ оказался длиннее, чем путь не проходящий через $e(a_0, b_0)$, то кратчайший путь между a_{i+1} и b_j не может иметь вида (3), так как в этом случае кратчайший путь между a_{i+1} и b_j содержал бы не кратчайший подпуть $[a_i, \ldots, a_0, b_0, \ldots, b_j]$ между a_i и b_j, что нарушало бы свойство оптимальности кратчайшего пути.

Таким образом, если расстояние между данной a_i и некоторой b_j не было пересчитано, то не следует пересчитывать и расстояния между b_j и всеми вершинами исходящими из a_i в сторону от корня дерева t_A вплоть до листовых вершин. На основании этого правила можно уменьшить размеры деревьев t_A и t_B, исключая вершины дерева t_A, которым не следует пересчитывать расстояния до b_0 и вершины дерева t_B, которым не следует пересчитывать расстояния до a_0. Формально в t_A и t_B включаются,

соответственно, вершины a_i и b_j, удовлетворяющие условиям

$$l(a_i, a_0) + w(a_0, b_0) < l(a_i, b_0),$$
$$l(b_j, b_0) + w(a_0, b_0) < l(b_j, a_0). \quad (4)$$

Схематическое изображение деревьев t_A и t_B представлено на рис. 1. Описание алгоритма пересчета расстояний при добавлении ребра дано в листингах Алгоритм 1, Алгоритм 2. Алгоритмы в данных листингах предполагают, что текущие расстояния между всеми парами вершин графа до добавления ребра $e(a_0, b_0)$ известны.

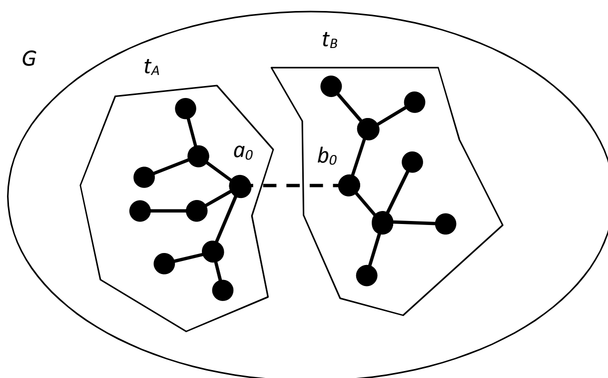

Рис. 1. Деревья t_A, t_B, между парами вершин которых при добавлении ребра $e(a_0, b_0)$ пересчитываются расстояния

Алгоритм 1 (Построение деревьев кратчайших путей).
`ПостроитьДеревоДобавления` (G, l, v_1, v_2)

1. t = Дерево(Узел(v_1)) *// дерево с корнем с индексом v_1*
2. q = Очередь(t.Корень()) *// q – структура типа FIFO*
3. d = Множество(v_1) *// d – мн-во посещенных вершин*
4. **Пока** q.Размер() > 0
5. v = q.Выбрать()
6.   **Для всех** ребер $e(v, u) \in G.E()$
7.   **Если** d.Содержит(u), **то**
8.   **Продолжить** *// continue*
9.   **Если** $l(v_1, u) = l(v_1, v) + G.w(v, u)$, **то**

10. d.Добавить(u)
11. **Если** $l(u, v_1) + G.w(v_1, v_2) < l(u, v_2)$, **то**
12. $c =$ Узел(u)
13. n.ДобавитьПотомка(c)
14. q.Добавить(c)
15. **Вернуть** t

Алгоритм 2 (Пересчет l при добавлении $e(a_0, b_0)$ веса w_{ab}).

`ДобавитьРебро` (G, l, a_0, b_0, w_{ab})

1. $G.E$.Добавить($e(a_0, b_0)$)
2. $G.w(a_0, b_0) = w_{ab}$
3. $G.w(b_0, a_0) = w_{ab}$
4. **Если** $l(a_0, b_0) \leqslant w_{ab}$, **то**
5. **Вернуть**
6. $t_A =$ ПостроитьДеревоДобавления(G, l, a_0, b_0)
7. $t_B =$ ПостроитьДеревоДобавления(G, l, b_0, a_0)
8. $q_a =$ Очередь(t_A.Корень())
9. **Пока** q_a.Размер() > 0
10. $a_i = q_a$.Выбрать()
11. $q_b =$ Очередь(t_B.Корень())
12. **Пока** q_b.Размер() > 0
13. $b_j = q_b$.Выбрать()
14. $l = l(a_i, a_0) + G.w(a_0, b_0) + l(b_0, b_j)$
15. **Если** $l < l(a_i, b_j)$, **то**
16. $l(a_i, b_j) = l$
17. $l(b_j, a_i) = l$
18. **Для** $k = 1$ **до** b_j.ЧислоПотомков()
19. q_b.Добавить(b_j.Потомок(k))
20. **Для** $k = 1$ **до** a_i.ЧислоПотомков()
21. q_a.Добавить(a_i.Потомок(k))

Алгоритм 1 по сути является проходом по графу поиском в ширину, поэтому его временная сложность соответствуют таковой поиска в ширину и равна $O(|V| + |E|)$, пространственная сложность равна $O(|V|^2 + |E|)$ из-за необходимости хранить расстояния. Если в алгоритме 2 условие в строке 4 не выполняется, то его временная сложность рассчитывается следующим образом.

Временная сложность сложность строк 6, 7 равна $O(|V| + |E|)$. Выбор и добавление вершин в очереди в строках 10, 13, 19 и 21 производится не больше $|V|$ раз, что с учетом вложенности цикла в строках 12-19 дает временную сложность выполнения строк 9–21 равную $O(|V|^2)$. То есть алгоритм 2 имеет временную сложность $O(|V|^2 + |E|)$. Пространственная сложность алгоритма 2 доминируется пространственной сложностью алгоритма 1 и равна $O(|V|^2 + |E|)$.

5. Алгоритм поиска кратчайших расстояний путем добавления ребер к остовному дереву

Алгоритм пересчета расстояний при добавлении ребра может использоваться для определения расстояний между всеми парами вершина графа (задача APSD). Для этого в графе выбирается произвольное остовное дерево t_f, для которого поиском в ширину находятся расстояния между всеми вершинами. После чего по очереди добавляются $|E| - (|V| - 1)$ не вошедших в t_f ребер графа, и для каждого добавленного ребра выполняется пересчет расстояний.

Алгоритм 3 (Расчет расстояний для дерева)**.**

`ВычислитьРасстоянияДерева` (t_f, l)

1. q = Очередь(t_f.Корень())
2. d = Множество(t_f.Корень())
3. **Пока** q.Размер() > 0
4. $v_j = q$.Выбрать()
5.   **Для** $m = 1$ **до** n.ЧислоПотомков()
6. $v_i = v_j$.Потомок(m)
7.   **Для всех** $v_k \in d$
8. $l(v_k, v_i) = l(v_k, v_j) + w(v_j, v_i)$
9. $l(v_i, v_k) = l(v_k, v_i)$
10. d.Добавить(v_i)
11. q.Добавить(v_i)

При обходе дерева t_f расстояния от каждой новой посещаемой вершины v_i до уже посещенных v_k считаются через роди-

тельскую вершину v_j вершины v_i:

$$l(v_k, v_i) = l(v_k, v_j) + w(v_j, v_i).$$

В разделе 7 приведены результаты вычислительного эксперимента, в котором сравнивается время работы алгоритма добавления ребра и скорость выполнения известных алгоритмов поиска кратчайших расстояний. Согласно результатам, можно сделать вывод: использование приведенного алгоритма будет эффективно только при очень малом числе добавляемых ребер. Так как скорость добавления каждого ребра увеличивается пропорционально квадрату числа вершин, как и в других методах поиска кратчайших расстояний, то эффективность для широкого диапазона числа вершин будет ограничена некоторым абсолютным значением. Если опираться на результаты в таблицах 1 и 3, то можно сделать вывод, что алгоритм добавления ребер к дереву быстрее алгоритма Дейкстры, если добавляемых ребер не больше сотни, и быстрее алгоритма разборки-сборки, если добавляемых ребер не больше нескольких десятков.

Описание алгоритма определения расстояний между всеми парами вершин графа с использованием алгоритма пересчета расстояний при добавлении ребра приведено в листингах Алгоритм 3, Алгоритм 4.

Алгоритм 4 (Расчет расстояний между всеми вершинами).

`ВычислитьРасстояния` (G, l)

1. u = Очередь() // *содержит не посещенные ребра*
2. t_f = ВыбратьДерево(G, u) // *u заполняется*
3. ВычислитьРасстоянияДерева (t_f, l)
4. **Пока** u.Размер() > 0
5. $e = u$.Выбрать()
6. $a_0 = e$.Вершина1()
7. $b_0 = e$.Вершина2()
8. $w_{ab} = e$.Вес()
9. ДобавитьРебро (G, l, a_0, b_0, w_{ab})

В алгоритме 3 цикл в строках 3–11 выполняется $|V|$ раз – по разу для каждой вершины дерева. Вложенный цикл в строках

7–9 выполняется в худшем случае $|V|-1$ раз. Таким образом, временная сложность алгоритма 3 равна $O(|V|^2)$. Однако время работы данного алгоритма может быть улучшено до линейного с помощью нахождения всех наименьших общих предков дерева [13]. В алгоритме 4 строка 2 выполняется за $O(|V|+|E|)$ (поиск в ширину), строка 3 за $O(|V|^2)$, цикл в строках 4–9 за $O(|E||V|^2+|E|^2)$. Таким образом временная сложность алгоритма 4 равна $O(|E||V|^2+|E|^2)$, пространственная сложность определяется строкой 9 и равна $O(|V|^2+|E|)$.

6. Удаление ребра и равноудаленные точки

Рассматривается удаление ребра $e(a_0,b_0)$ веса $w(a_0,b_0)$ между вершинами a_0 и b_0. Необходимо пересчитать расстояния $l(a_i,b_j)$, которые в результате удаления данного ребра увеличатся. Предполагается, что вес удаляемого ребра равен текущему расстоянию между вершинами $w(a_0,b_0)=l(a_0,b_0)$, так как в противном случае пересчет расстояний производить не нужно.

Как и в случае с добавлением ребра, используя выражения (1), множество вершин графа V можно поделить на множества A и B. На этих множествах так же можно построить деревья кратчайших путей t_A и t_B с корнями a_0 и b_0. Эти деревья будут обладать тем же полезным свойством: если расстояние между a_i и b_j не следует пересчитывать, то не следует пересчитывать и расстояния между всеми потомками a_i и b_j. Размеры деревьев t_A и t_B можно уменьшить, но для этого вместо неравенств (4) следует использовать следующие равенства:

$$\begin{aligned} l(a_i,a_0)+w(a_0,b_0)&=l(a_i,b_0),\\ l(b_j,b_0)+w(a_0,b_0)&=l(b_j,a_0). \end{aligned} \tag{5}$$

Равенства (5) оставляют в деревьях t_A и t_B только те вершины a_i и b_j, для которых существует кратчайший путь до, соответственно, вершин b_0 и a_0, проходящий через ребро $e(a_0,b_0)$. Так как в графе между любой парой вершин может существовать несколько кратчайший путей, не обязательно, что между вершинами деревьев t_A и t_B, полученных таким образом, произойдет хотя бы один пересчет расстояний.

Существенным отличием процедуры удаления ребра от случая добавления ребра является отсутствие информации о длине кратчайшего альтернативного пути. При добавлении ребра для пересчета расстояния между вершинами a_i и b_j используется неравенство (2), в котором длина известного (до добавления ребра $e(a_0, b_0)$) кратчайшего альтернативного пути $l(a_i, b_j)$ сравнивается с длиной пути через ребро $e(a_0, b_0)$. При удалении ребра известно лишь расстояние между a_i и b_j – длина кратчайшего пути (возможно проходящего через $e(a_0, b_0)$), второй по длине альтернативный путь между a_i и b_j неизвестен. Таким образом, возникает задача просмотра альтернативных путей и выбора из них наименьшего по длине.

На каждом из альтернативных путей между вершинами a_0 и b_0 существует точка, от которой расстояние до a_0 и до b_0 одинаково. Будем называть такие точки *равноудаленными точками* (РУТ). Используя РУТ можно обозначить все возможные варианты альтернативных путей между a_0 и b_0, ставя в соответствие каждому варианту отличную РУТ. Набор РУТ также определит и все возможные варианты альтернативных путей, не проходящих через ребро $e(a_0, b_0)$, между всеми парами вершин деревьев t_A и t_B. Действительно, пути между вершинами a_i и b_j через РУТ представляют собой пути, в которых вместо ребра $e(a_0, b_0)$ содержится подпуть через РУТ. Будем обозначать РУТ $C = \{c_k\}$, рис. 2 представляет схематическое изображение множества РУТ.

Существует 2 варианта расположения РУТ. Во-первых, РУТ c_k, находящиеся в вершинах графа. Для таких справедливо равенство

$$l(c_k, a_0) = l(c_k, b_0). \tag{6}$$

Во-вторых, РУТ, находящиеся на ребрах. Такие РУТ находятся на ребрах $e(v, u)$, удовлетворяющим всем трем неравенствам

$$\begin{gathered} |l(u, b_0) - l(v, a_0)| < w(v, u), \\ l(v, a_0) < l(v, b_0), \\ l(u, b_0) < l(u, a_0). \end{gathered} \tag{7}$$

Расстояния между РУТ на ребрах c_k и некоторой вершиной z

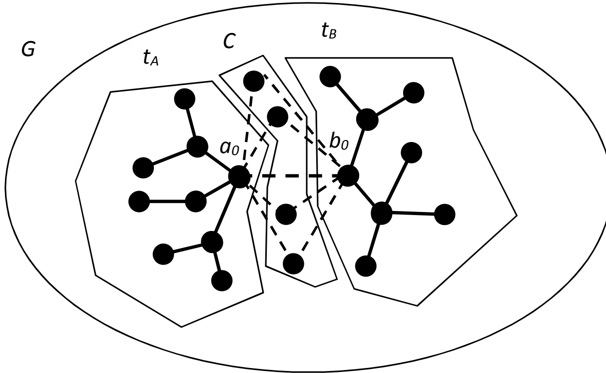

Рис. 2. Множество равноудаленных точек (РУТ) C при удалении ребра $e(a_0, b_0)$

определяются формуле

$$l(z, c_k) = \begin{cases} l(z, v) + w'(v, c_k), & \text{если } l(z, a_0) \leqslant l(z, b_0); \\ l(z, u) + w'(u, c_k) & \text{иначе.} \end{cases} \quad (8)$$

где $w'(v, c_k)$ и $w'(u, c_k)$ – длины отрезков ребра $e(v, u)$, определяемые по формулам

$$w'(v, c_k) = (l(u, b_0) + w(v, u) - l(v, a_0))/2, \\ w'(u, c_k) = w(v, u) - w'(v, c_k). \quad (9)$$

Так как РУТ имеют одинаковое расстояние до вершин, инцидентных удаляемому ребру, то при удалении ребра $e(a_0, b_0)$ расстояния от РУТ до всех остальных вершин графа не изменится. Действительно, если бы расстояние между РУТ c_k и некоторой вершиной z после удаления ребра $e(a_0, b_0)$ изменилось, то это значило бы, что до удаления ребра кратчайший путь между z и c_k имел вид $[c_k, \ldots, a_0, b_0, \ldots, z]$, что невозможно так как наличие не кратчайшего подпути $[c_k, \ldots, a_0, b_0]$ противоречило бы свойству оптимальности кратчайшего пути. Таким образом, новое расстояние между вершинами a_i и b_j деревьев t_A и t_B может быть определено как минимальная сумма расстояний через РУТ: $l(a_i, b_j) = \min\limits_k (l(a_i, c_k) + l(c_k, b_j))$. Можно уменьшить число «холостых» проходов по деревьям t_A и t_B (без фактического

пересчета расстояний), если использовать не расстояния до РУТ, а приращения (увеличения) расстояний через РУТ.

Будем обозначать $s(a_i, b_j)$ и называть *приращением расстояния* $l(a_i, b_j)$ увеличение расстояния $l(a_i, b_j)$ после удаления ребра $e(a_0, b_0)$. Тогда новое расстояние между a_i и b_j после удаления $e(a_0, b_0)$ определяется по формуле

$$l(a_i, b_j) = l(a_i, b_j) + s(a_i, b_j).$$

Будем обозначать $s^k(a_i, b_0)$, $s^k(b_j, a_0)$ и называть *приращениями расстояний* $l(a_i, b_0)$, $l(b_j, a_0)$ *через РУТ* c_k увеличение длин путей между соответствующими вершинами при прокладывании их через РУТ c_k при удалении ребра $e(a_0, b_0)$.

Для вершин, инцидентных удаляемому ребру, приращение определяется просто как минимальное увеличение пути через все существующие РУТ:

$$s(a_0, b_0) = \min_k s^k(a_0, b_0), \tag{10}$$

$$s^k(a_0, b_0) = l(a_0, c_k) + l(c_k, b_0) - w(a_0, b_0). \tag{11}$$

Для отличных от a_0,b_0 вершин деревьев t_A и t_B приращения через РУТ определяются как изменения приращений через РУТ относительно родительского узла в дереве кратчайших путей:

$$\begin{aligned} s^k(a_{i+1}, b_0) &= s^k(a_i, b_0) - w(a_{i+1}, a_i) + l(a_{i+1}, c_k) - l(a_i, c_k), \\ s^k(b_{j+1}, a_0) &= s^k(b_j, a_0) - w(b_{j+1}, b_j) + l(b_{j+1}, c_k) - l(b_i, c_k), \end{aligned} \tag{12}$$

где a_{i+1} и b_{j+1} – прямые потомки узлов a_i и b_j, соответственно, в деревьях t_A и t_B. Справедливость первой из формул (12) показывается следующим образом. По определению, приращение $s^k(a_{i+1}, b_0)$ есть разность между расстоянием $l(a_{i+1}, b_0)$ через вершину c^k после удаления ребра $e(a_0, b_0)$ и расстоянием $l(a_{i+1}, b_0)$ до удаления ребра $e(a_0, b_0)$. Расстояния $l(a_{i+1}, b_0)$ и $l(a_i, b_0)$ через вершину c^k после удаления ребра $e(a_0, b_0)$ равны, соответственно, $l(a_{i+1}, c_k) + l(c_k, b_0)$ и $l(a_i, c_k) + l(c_k, b_0)$. Согласно построению дерева t_A (Алгоритм 1 и (5)), до удаления ребра $e(a_0, b_0)$ справедливо $l(a_{i+1}, b_0) = l(a_i, b_0) + w(a_{i+1}, a_i)$. Таким образом, получаем $s^k(a_{i+1}, b_0) = l(a_{i+1}, c_k) + l(c_k, b_0) - [l(a_i, b_0) + w(a_{i+1}, a_i)] = l(a_{i+1}, c_k) + l(c_k, b_0) - l(a_i, b_0) - w(a_{i+1}, a_i) +$

$l(a_i, c_k) - l(a_i, c_k) = s^k(a_i, b_0) - w(a_{i+1}, a_i) + l(a_{i+1}, c_k) - l(a_i, c_k)$. Аналогично для второй из формул (12).

Приращения расстояний для отличных от a_0, b_0 вершин a_i и b_j деревьев t_A и t_B определяются как сумма приращений через РУТ от a_i и от b_j с вычетом приращения через РУТ между вершинами удаляемого ребра, так как оно входит в сумму дважды:

$$s(a_i, b_j) = \min_k (s^k(a_i, b_0) + s^k(b_j, a_0) - s^k(a_0, b_0)). \quad (13)$$

Действительно, согласно построению дерева t_A, определению приращения через РУТ, формуле (11) и симметричности весовой функции графа: $s^k(a_i, b_j) = l(a_i, c_k) + l(c_k, b_j) - \big[l(a_i, a_0) + w(a_0, b_0) + l(b_0, b_j)\big] = \big[l(a_i, c_k) + l(c_k, b_0) - l(a_i, a_0) - w(a_0, b_0)\big] + \big[l(b_j, c_k) + l(c_k, a_0) - l(b_j, b_0) - w(b_0, a_0)\big] - \big[l(a_0, c_k) + l(c_k, b_0) - w(a_0, b_0)\big] = s^k(a_i, b_0) + s^k(b_j, a_0) - s^k(a_0, b_0)$. Согласно определению множества РУТ, $s(a_i, b_j) = \min_k s^k(a_i, b_j)$, что доказывает корректность (13).

Если для вершины a_i дерева t_A существует нулевое приращение расстояния через РУТ, т.е. $\min_k s^k(a_i, b_0) = 0$, то для a_i и всех ее потомков в дереве t_A вплоть до листового уровня не нужно производить пересчет расстояний ни до одной вершины b_j дерева t_B. Действительно, в этом случае наличие нулевого приращения расстояния через РУТ означает существование альтернативного пути между a_i и b_0, не проходящего через удаляемое ребро $e(a_0, b_0)$, и имеющего такую же длину, как и путь через $e(a_0, b_0)$. Таким образом, расстояние $l(a_i, b_0)$ пересчитывать не нужно – оно не изменится. Так как t_A и t_B деревья кратчайших путей, кратчайший путь между любым a_x потомком a_i и любым b_y потомком b_0 содержит до удаления ребра $e(a_0, b_0)$ подпуть вида $[a_i, \ldots, b_0]$, так как длина $l(a_i, b_0)$ этого подпути не изменилась, то не изменится и расстояние $l(a_x, b_y)$. Аналогичное справедливо и для вершины b_j дерева t_B и ее потомков в случае $\min_k s^k(b_j, a_0) = 0$.

Можно сократить и само число РУТ, используя два правила:

1. Из C можно исключить те РУТ, находящиеся в вершинах, которые смежны только с вершинами, которые тоже являют-

ся РУТ в вершинах. В этом случае пути через исключаемые РУТ будут проходить также и через РУТ, которые будут не исключены.

2. Если для деревьев t_A и t_B известны вершины a_f и b_f, находящиеся на наибольшем расстоянии от удаляемого ребра $e(a_0, b_0)$, т.е. $l(a_f, a_0) = \max_i l(a_i, a_0)$ и $l(b_f, b_0) = \max_j l(b_j, b_0)$, а также ближайшая к удаляемому ребру РУТ c_n, т.е. $l(c_n, a_0) = \min_i l(c_i, a_0)$, то нужно оставить в C только те РУТ c_k, для которых верно

$$l(c_k, a_0) < l(a_f, a_0) + l(b_f, b_0) + l(c_n, a_0). \tag{14}$$

Действительно, предположим, что РУТ c_k нельзя удалять из множества C, так как после удаления ребра $e(a_0, b_0)$ кратчайший путь между некоторыми вершинами a_i и b_j, соответственно, деревьев t_A и t_B проходит через c_k, т.е. справедливо неравенство

$$l(a_i, c_k) + l(c_k, b_j) < l(a_i, c_n) + l(c_n, b_j) \tag{15}$$

и при этом для c_k не выполняется (14), т.е.

$$l(c_k, a_0) \geqslant l(a_f, a_0) + l(b_f, b_0) + l(c_n, a_0). \tag{16}$$

В силу неравенства треугольника, определения РУТ и симметричности весовой функции графа справедливы неравенства

$$\begin{gathered}
l(c_k, a_0) - l(a_i, a_0) \leqslant l(a_i, c_k), \\
l(c_k, a_0) - l(b_j, b_0) = l(c_k, b_0) - l(b_j, b_0) \leqslant l(c_k, b_j), \\
l(a_i, c_n) \leqslant l(a_i, a_0) + l(c_n, a_0), \\
l(c_n, b_j) \leqslant l(b_j, b_0) + l(b_0, c_n) = l(b_j, b_0) + l(c_n, a_0).
\end{gathered}$$

Подставляя данные неравенства в (15), получаем $2 \cdot l(c_k, a_0) - l(a_i, a_0) - l(b_j, b_0) < 2 \cdot l(c_n, a_0) + l(a_i, a_0) + l(b_j, b_0)$. Используя (16), последнее неравенство эквивалентно $2 \cdot \big[l(a_f, a_0) + l(b_f, b_0) + l(c_n, a_0)\big] - l(a_i, a_0) - l(b_j, b_0) < 2 \cdot l(c_n, a_0) + l(a_i, a_0) + l(b_j, b_0)$, что эквивалентно $2 \cdot \big[l(a_f, a_0) + l(b_f, b_0)\big] < 2 \cdot \big[l(a_i, a_0) + l(b_j, b_0)\big]$. Данное неравенство противоречит определению b_f и a_f, что доказывает корректность правила $r2$.

Описание алгоритма пересчета расстояний при удалении ребра приведено в листинге Алгоритм 5.

Алгоритм 5 (Пересчет l при удалении $e(a_0, b_0)$ веса w_{ab}).

`УдалитьРебро` (G, l, a_0, b_0, w_{ab})

1. $G.E$.Удалить($e(a_0, b_0)$)
2. $G.w(a_0, b_0) = \infty$
3. $G.w(b_0, a_0) = \infty$
4. **Если** $l(a_0, b_0) < w_{ab}$, **то**
5.   **Вернуть**
6. $t_A =$ ПостроитьДеревоУдаления(G, l, a_0, b_0)
7. $t_B =$ ПостроитьДеревоУдаления(G, l, b_0, a_0)
   // *ПостроитьДеревоДобавления с* (5) *вместо* (4)
8. $c = \{$.НайтиРУТ.$(G, l)\}$ // *формулы (6, 7), правила* $r1$, $r2$
9. $q_a =$ Очередь(t_A.Корень())
10. **Пока** q_a.Размер() > 0
11. $a_i = q_a$.Выбрать()
12. $S_k = \{$.НайтиПриращениеЧерезРут.$(G, l, c, a_i)\}$
        // *формулы (11, 12, 8, 9)*
13.   **Если** $\exists s^k \in S_k : s^k > 0$, **то**
14. $q_b =$ Очередь(t_B.Корень())
15.   **Пока** q_b.Размер() > 0
16. $b_j = q_b$.Выбрать()
17. $s =$ НайтиПриращение(G, l, c, a_i, b_j)
                // *формулы (10, 13)*
18.   **Если** $s > 0$, **то**
19. $l(a_i, b_j) = l(a_i, b_j) + s$
20. $l(b_j, a_i) = l(b_j, a_i) + s$
21.   **Для** $k = 1$ **до** b_j.ЧислоПотомков()
22. q_b.Добавить(b_j.Потомок(k))
23.   **Для** $k = 1$ **до** a_i.ЧислоПотомков()
24. q_a.Добавить(a_i.Потомок(k))

Если в алгоритме 5 условие в строке 4 не выполняется, то его временная сложность рассчитывается следующим образом. Строки 6, 7, 8 выполняются за $O(|V|+|E|)$. Так как строки 12, 17 выполняются за $O(|V|)$ временная сложность строк 10–24 определяется суммарной временной сложностью дважды вложенной строки 17 и равняется $O(|V|^3)$. Таким образом, временная слож-

ность алгоритма 5 равна $O(|V|^3)$, пространственная сложность, как и в предыдущих алгоритмах, равна $O(|V|^2 + |E|)$.

7. Вычислительный эксперимент

Практическая эффективность предложенного метода актуализации расстояний в динамическом графе была оценена с помощью вычислительного эксперимента. В качестве тестовых данных использовались случайные связные подграфы заданной размерности графов автомобильных дорог европейских стран из проекта OpenStreetMap [20]. Эксперимент состоял в удалении случайного ребра, которое не нарушает связности графа, и последующем добавлении этого ребра обратно к графу, при этом фиксировалось время актуализации расстояний после каждой операции. Данная процедура повторялась 100 раз, после чего определялось среднее время актуализации расстояний каждым из сравниваемых алгоритмов. Исходный код предложенных в статье алгоритмов (ДУТ), алгоритма Дейкстры (СД) и алгоритма разборки-сборки графа (СРС) был написан на языке C++, код остальных тестируемых в статье алгоритмов был реализован на языке C. При компиляции использовался 64-битный компилятор MS Visual Studio 2013 с флагом оптимизации O2. Тесты проводились на ЭВМ с процессором Intel Core i5-4690K и объемом оперативной памяти 24 Гбайт.

Предложенные алгоритмы пересчета расстояний (ДУТ) при добавлении ребра (уменьшении веса ребра) и удалении ребра (увеличение веса ребра) сравнивались с решением задачи статическими алгоритмами – полным пересчетом расстояний: n-кратным (по числу вершин графа) запуском алгоритма Дейкстры (СД); алгоритмом разборки и сборки графа ([2], СРС); основанном на алгоритме Дейкстры методом, просматривающем только ребра, содержащиеся в локально кратчайших путях ([6], СЛКП). Также было произведено тестирование быстрых динамических алгоритмов [6]: адаптации алгоритма Кинга ([17], ДК) без ограничения на максимальный вес ребра в графе C с временной и пространственной сложностями $\tilde{O}(n^{2,5}\sqrt{C})$; алгоритма Рамалин-

Таблица 1. Характеристики тестовых графов и среднее время актуализации расстояний статическими алгоритмами

Группа графов (число графов)	Среднее число вершин	Средняя степень	СД, с	СРС, с	СЛКП, с
G1(10)	10^3	2,19	0,059	0,013	0,369
G2(10)	$2 \cdot 10^3$	2,22	0,259	0,046	2,26
G3(10)	$3 \cdot 10^3$	2,23	0,615	0,122	6,07
G4(10)	$4 \cdot 10^3$	2,2	1,1	0,253	12,5
G5(10)	$5 \cdot 10^3$	2,21	1,8	0,376	24,5
G6(10)	$6 \cdot 10^3$	2,21	2,56	0,552	37,8
G7(10)	$7 \cdot 10^3$	2,23	3,63	0,78	61,2
G8(10)	$8 \cdot 10^3$	2,21	4,66	1,08	80,1
G9(10)	$9 \cdot 10^3$	2,22	5,98	1,38	109
G10(10)	10^4	2,2	7,52	1,85	142

Таблица 2. Среднее время (в секундах) актуализации расстояний динамическими алгоритмами после удаления ребра

Группа графов	ДУТ, с	ДК, с	ДРР, с	ДДИ, с
G1	$5{,}8 \cdot 10^{-4}$	0,325	0,009	0,012
G2	0,003	2	0,027	0,045
G3	0,005	–	0,051	0,104
G4	0,011	–	0,099	0,24
G5	0,016	–	0,108	0,24
G6	0,02	–	0,154	–
G7	0,03	–	0,22	–
G8	0,038	–	0,302	–
G9	0,05	–	0,333	–
G10	0,07	–	0,423	–

гама и Репса ([22], ДРР), использующего при пересчете хранимую информацию о кратчайших путях, с временной сложностью $O(mn{+}n^2 logn)$ и пространственной сложностью $O(n^2)$; алгоритма Деметреску и Италиано ([7], ДДИ), пользующегося при актуализации расстояний исторически локальными путями с временной сложностью $\tilde{O}(n^2)$ и пространственной сложностью $\tilde{O}(mn)$.

Таблица 3. Среднее время (в секундах) актуализации расстояний динамическими алгоритмами после добавления ребра

Группа графов	ДУТ, с	ДК, с	ДРР, с	ДДИ, с
G1	$3{,}7 \cdot 10^{-4}$	$1{,}8 \cdot 10^{-4}$	0,009	0,017
G2	0,001	$1{,}7 \cdot 10^{-4}$	0,023	0,065
G3	0,003	–	0,044	0,148
G4	0,005	–	0,087	0,343
G5	0,006	–	0,087	0,426
G6	0,008	–	0,129	–
G7	0,012	–	0,183	–
G8	0,015	–	0,253	–
G9	0,02	–	0,272	–
G10	0,029	–	0,349	–

В эксперименте использовалась реализация алгоритма Дейкстры с двоичной кучей из библиотеки Boost Graph Library [25]. Алгоритмы СЛКП, ДК, ДРР и ДДИ использовались в реализации [10]. В тех случаях, когда алгоритмы в реализации [10] не смогли завершить вычисления из-за ошибок в авторском коде, в таблицах стоит прочерк. Характеристики тестовых графов и среднее время актуализации расстояний статическими алгоритмами представлены в таблице 1. Среднее время актуализации расстояний динамическими алгоритмами после удаления и добавления ребра представлены в таблицах 2 и 3 соответственно. Таблица 4 содержит данные о среднем объеме используемой оперативной памяти динамическими алгоритмами. В таблице 5 содержатся дополнительные данные о результатах тестирования разработанного алгоритма ДУТ.

Таблица 4. Средний объем используемой оперативной памяти динамическими алгоритмами

Группа графов	ДУТ, с	ДК, с	ДРР, с	ДДИ, с
G1	5,36 Мбайт	504 Мбайт	155 Мбайт	147 Мбайт
G2	18,7 Мбайт	2,77 Гбайт	618 Мбайт	592 Мбайт
G3	39,2 Мбайт	–	1,35 Гбайт	1,29 Гбайт
G4	70,6 Мбайт	–	2,4 Гбайт	2,28 Гбайт
G5	104 Мбайт	–	3,79 Гбайт	3,45 Гбайт
G6	147 Мбайт	–	5,45 Гбайт	–
G7	200 Мбайт	–	7,36 Гбайт	–
G8	261 Мбайт	–	9,69 Гбайт	–
G9	329 Мбайт	–	12,2 Гбайт	–
G10	406 Мбайт	–	14,8 Гбайт	–

Полученные результаты свидетельствуют о том, что предложенные алгоритмы позволяют существенно сократить время актуализации расстояний после удаления ребра в сравнении с рассмотренными алгоритмами. Так, например, для графов размерности 10^4 время счета меньше в среднем в 6 раз. Быстрейший пересчет расстояний после добавления ребра производит алгоритм ДК, однако из-за ошибок в авторском коде [10] не ясна его скорость на графах размерности больше $2 \cdot 10^3$. Еще одним важным практическим преимуществом предложенных алгоритмов является использование гораздо меньшего объема оперативной памяти в сравнении с другими динамическими алгоритмами.

8. Заключение

На практике взвешенные графы большого размера моделируют такие объекты (например, дорожные сети), для которых крайне маловероятно одновременное изменение весов у большого числа ребер. Скорее, наоборот – изменение ситуации (дорожной обстановки) должно сопровождаться постоянным и очень быстрым процессом пересчета расстояний между вершинами при

Таблица 5. Результаты тестирования алгоритма ДУТ

Группа графов	Добавление. Макс. время, с	Удаление. Макс. время, с	Среднее число РУТ	Макс. число РУТ
G1	0,016	0,016	4,94	29
G2	0,016	0,016	6,66	42
G3	0,031	0,047	9,77	61
G4	0,047	0,14	8,83	58
G5	0,062	0,234	11,5	54
G6	0,078	0,218	10,3	60
G7	0,187	0,343	16,3	78
G8	0,209	0,608	11,5	68
G9	0,211	0,64	15,2	102
G10	0,281	0,904	14,3	85

изменении весов отдельных ребер.

Приведенный динамический алгоритм коррекции кратчайших расстояний позволяет поддерживать актуальные расстояния между вершинами в режиме реального времени даже для графов большой размерности.

Литература

1. РОДИОНОВ В.В. *Параметрическая задача о кратчайших расстояниях* // Ж. вычисл. матем. и матем. физ. – 1968. – Т. 8, №5. – С. 1173–1177.
2. УРАКОВ А.Р., ТИМЕРЯЕВ Т.В. *Алгоритм поиска кратчайших путей для разреженных графов большой размерности* // Прикладная дискретная математика. – 2013. – №1(19). – С. 84–92.
3. AUSIELLO G., ITALIANO G.F., MARCHETTI-SPACCAMELA A., NANNI U. *Incremental algorithms for minimal length paths* // Journal of Algorithms. – 2004. – Vol. 12, №4. – P. 615–638.

4. BASWANA S., HARIHARAN R., SEN S. *Improved decremental algorithms for maintaining transitive closure and all-pairs shortest paths* // Journal of Algorithms. – 2007. – Vol. 62, №2. – P. 74–92.
5. CICERONE S., DI STEFANO G., FRIGIONI D., NANNI U. *A fully dynamic algorithm for distributed shortest paths* // Theoretical Computer Science. – 2003. – Vol. 297, №1–3. – P. 83–102.
6. DEMETRESCU C., EMILIOZZI S., ITALIANO G.F. *Experimental analysis of dynamic all pairs shortest path algorithms* // Proc. of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms. – Philadelphia: Society for Industrial and Applied Mathematics, 2004. – P. 362–371.
7. DEMETRESCU C., ITALIANO G.F. *A new approach to dynamic all pairs shortest paths* // Journal of the ACM. – 2004. – Vol. 51, №6. – P. 968–992.
8. DIONNE R. *Etude et extension d'un algorithme de Murchland* // INFOR. – 1978. – №169. – P. 132–146.
9. EVEN S., GAZIT H. *Updating distances in dynamic graphs* // Methods of Operations Research. – 1985. – №49. – P. 371–387.
10. *Experimental Evaluation of Dynamic All Pairs Shortest Path Algorithms* [Электронный ресурс]. – URL: http://www.dis.uniroma1.it/~demetres/experim/dsp/ (дата обращения: 27.12.2016).
11. FAKCHAROEMPHOL J., RAO S. *Planar graphs, negative weight edges, shortest paths, and near linear time* // Journal of Computer and System Sciences. – 2006. – Vol. 72, №5. – P. 868–889.
12. FRIGIONI D., MARCHETTI-SPACCAMELA A., NANNI U. *Fully dynamic algorithms for maintaining shortest paths trees* // Journal of Algorithms. – 2000. – №34. – P. 351–381.

13. GABOW H., TARJAN R. *A linear-time algorithm for a special case of disjoint set union* // Journal of Computer and System Sciences. – 1985. – Vol. 30, №2. – P. 209–221.
14. GRECO S., MOLINARO C., PULICE C. *Fully dynamic algorithms for maintaining shortest paths trees* // Proc. of the 28th International Conference on Scientific and Statistical Database Management. – New York: ACM, 2016. – P. 9:1–9:12.
15. HENZINGER M., KLEIN P., NANONGKAI D. *Dynamic Approximate All-Pairs Shortest Paths: Breaking the O(mn) Barrier and Derandomization* // Proc. of the 2013 IEEE 54th Annual Symposium on Foundations of Computer Science. – Washington: IEEE Computer Society, 2013. – P. 538–547.
16. HENZINGER M., KLEIN P., RAO S., SUBRAMANIAN S. *Faster shortest-path algorithms for planar graphs* // Journal of Computer and System Sciences. – 1997. – Vol. 55, №1. – P. 3–23.
17. KING V. *Fully dynamic algorithms for maintaining all–pairs shortest paths and transitiveclosure in digraphs* // Proc. of the 40th Annual Symposium on Foundations of Computer Science. – Washington: IEEE Computer Society, 1999. – P. 81–99.
18. LOUBAL P. *A network evaluation procedure* // Highway Research Record. – 1967. – №205. – P. 96–109.
19. MURCHLAND J. *The effect of increasing or decreasing the length of a single arc on all shortest distances in a graph* // Technical report, LBS-TNT-26. – London: London Business School, Transport Network Theory Unit, 1967.
20. *OpenStreetMap* [Электронный ресурс]. – URL: http://wiki.openstreetmap.org/wiki/Planet.osm (дата обращения: 27.12.2016).

21. PANG C., DONG G., RAMAMOHANARAO K. *Incremental maintenance of shortest distance and transitive closure in first-order logic and SQL* // ACM Transactions on Database Systems. – 2005. – Vol. 30, №3. – P. 698–721.
22. RAMALINGAM G., REPS T. *An incremental algorithm for a generalization of the shortest path problem* // Journal of Algorithms. – 1996. – №21. – P. 267–305.
23. RAMARAO K., VENKATESAN S. *On finding and updating shortest paths distributively* // Journal of Algorithms. – 1992. – Vol. 13, №2. – P. 235–257.
24. RODITTY L., ZWICK U. *Dynamic Approximate All-Pairs Shortest Paths in Undirected Graphs* // SIAM Journal on Computing. – 2012. – Vol. 41, №3. – P. 670–683.
25. *The Boost Graph Library* [Электронный ресурс]. – URL: www.boost.org/doc/libs/1_59_0/libs/graph/doc/index.html (дата обращения: 27.12.2016).
26. THORUP M. *Fully-Dynamic All-Pairs Shortest Paths: Faster and Allowing Negative Cycles* // Proc. of the 9th Scandinavian Workshop on Algorithm Theory. – Berlin: Springer, 2004. – P. 384–396.

ALGORITHM FOR DYNAMIC ALL-PAIRS DISTANCES IN GRAPH

Airat Urakov, Ufa State Aviation Technical University, Ufa, Candidate of Sciences, docent (urakov@ufanet.ru).
Timofey Timeryaev, Ufa State Aviation Technical University, Ufa (timeryaev@yandex.ru).

Abstract: Fully dynamic all-pairs graph distances problem for undirected graphs with positive edge weights is considered. Edge weights can change arbitrary so distances between vertices should be kept actual. We propose an algorithm taking into account all possible distance changes by the use of edge addition and deletion procedures. The developed algorithm uses the notion of points equidistant to the vertices incident to the edge being deleted for an edge deletion procedure. This allows to significantly reduce time and memory complexity of the graph distance actualization task in practical scenarios. The conducted computational experiments showed that the proposed algorithms outperforms the fastest known methods.

Keywords: dynamic graph distances, graph update, equidistant points, graph actualization.

Статья представлена к публикации членом редакционной коллегии О.П. Кузнецовым.

Поступила в редакцию 16.08.2016.
Дата опубликования 31.01.2017.