УДК 519.17

ИСПОЛЬЗОВАНИЕ СРЕДСТВ БАЗ ДАННЫХ ДЛЯ МАТЕМАТИЧЕСКИХ ВЫЧИСЛЕНИЙ НА ПРИМЕРЕ АЛГОРИТМА ДЕЙКСТРЫ

© В.Ю. Маркеев, А.А. Крючков

Ключевые слова: теория графов; задача о минимальном пути; алгоритм Дейкстры; базы данных; хранимые процедуры.

Рассмотрена реализация алгоритма Дейкстры средствами баз данных. Показаны преимущества данного подхода для задач большой размерности.

Рассмотрим задачу о нахождении *минимального пути* между заданными *вершинами* (*узлами*) v и w *взвешенного графа* G. Решение некоторых прикладных задач нередко базируется на поиске кратчайшего маршрута, соединяющего определенные вершины в заданном графе.

Впервые *задача о минимальном пути* была решена в 1959 г. норвежским программистом и математиком **Эдсгером Дейкстрой**. Он создал алгоритм поиска длины минимального пути для неотрицательных значений дуг графа в сети. Данный алгоритм основан на лемме о кратчайших цепях, которая рассматривается в [1].

Повторим несколько понятий теории графов. ***Цепью*** называется последовательность соединенных вершин. ***Длиной цепи*** в графе является сумма длин дуг, входящих в эту цепь. ***Путем*** из вершины v до вершины w называется цепь, соединяющая эти вершины. ***Длиной пути*** от v до w назовем длину цепи, соединяющей эти вершины.

В рамках данной задачи считается, что вес ребра (длина дуги) есть действительная величина, а длина пути от вершины до самой себя равна нулю.

Алгоритм Дейкстры. Шаг алгоритма заключается в последовательном рассмотрении непосещенных узлов с определенным расстоянием в порядке возрастания данного расстояния. Для каждой из непосещенных соседних (связаны ребром) вершин рассматриваемого узла проверяется, является ли путь до «соседа» через данный узел короче, чем ранее известное значение расстояния до «соседа». Путь до «соседа» через рассматриваемый узел равняется сумме длины ребра, соединяющего эти вершины, и значения расстояния до рассматриваемого узла. В случае если приведенное условие истинно, то следует заменить значение кратчайшего пути до «соседа» на более актуальное значение пути через рассматриваемый узел. По завершении проверки соседних вершин рассмотренный узел помечается посещенным, т. е. кратчайшее расстояние до него уже найдено и далее не участвует в расчете.

Шаг алгоритма повторяется до тех пор, пока не закончатся непосещенные вершины с определенным расстоянием.

На первом шаге известно только расстояние от заданной вершины до самой себя, которое равняется нулю. Этот узел и будет рассмотрен. Расстояния до других вершин еще не определены, что удобно выражать в эквиваленте бесконечности. Известные значения соседних вершин равняются бесконечности, что, разумеется, больше суммы длины ребра и нулевого значения заданного узла, поэтому значения расстояний до «соседей» будут обновлены. Данный узел помечается посещенным. Затем выполняется следующий шаг и т. д.

Реализация алгоритма. Исторически сложилось, что для математических вычислений в компьютерной среде в большинстве случаев используются такие языки программирования, как C++ (лидерство, в силу высокой скорости), Java (кроссплатформенность) и подобные. Что касается баз данных, то их принято использовать преимущественно для длительного хранения информации, а для серьезных вычислений они считаются непригодными в силу существенно невысокой скорости. Однако с увеличением объемов задач ситуация кардинально изменится.

Сначала рассмотрим классическую реализацию данного алгоритма посредством языка C++. Будем использовать для хранения информации о графе видоизмененную матрицу смежности, у которой на главной диагонали стоят нули, а на пересечении вершин длины ребер, соединяющих эти вершины. Отметим, что до изолированных вершин (вершин не инцидентных ни одному ребру) расстояние равно бесконечности, следовательно, на всех пересечениях изолированной вершины с другими в этой матрице поставим бесконечность (такую матрицу называют *матрицей весовых коэффициентов*). Как было показано выше, стандартная реализация основана на последовательном обходе массива и нахождении наименьших расстояний. Рассмотрим листинг ядра алгоритма (рис. 1).

Отметим, что в силу особенности языка C++ все вычисления и данные хранятся в оперативной памяти, чтобы повысить скорость вычислений, в то время как процессор используется в малом количестве. В дальнейшем увидим достоинства и некоторые недостатки данного подхода.

Алгоритм Дейкстры средствами баз данных. Для представления графа в базе данных будем использовать две связанные таблицы ***nodes*** (вершины) и ***edges*** (ребра). Таблица ***nodes*** имеет два поля: порядковый номер

вершины и ее имя. В свою очередь, в таблице ***edges*** содержится 4 поля: порядковый номер, начальная вершина, конечная вершина ребра и вес самого ребра. Заметим, что поля с начальными и конечными вершинами таблицы ***edges*** ссылаются на порядковый номер (первичный ключ) соответствующей записи (узла) таблицы ***nodes***. Структура организации базы данных проиллюстрирована на рис. 2.

Отдельно стоит сказать про таблицу ***ground*** (рабочая область), она имеет четыре поля: порядковый номер, номер вершины (ссылка на таблицу ***nodes***), длину минимального пути до этой вершины, которая будет уточняться в процессе вычислений, и признак посещенности. Изначально расстояние до всех вершин, кроме заданной, равно бесконечности (выразим это значением null), а все вершины считаются непосещенными. С каждым шагом алгоритма информация в данной таблице будет уточняться. Для эффективной обработки информации о графе, с точки зрения архитектуры баз данных, предпочтительно использовать список ребер вместо матрицы весовых коэффициентов, т. к. список ребер позволяет наиболее выгодным образом выбирать ребра по определенному запросу. Рассмотрим подробнее реализацию данного алгоритма на языке хранимых процедур СУБД MySQL (рис. 3).

Преимуществом данного подхода, помимо наглядности, является механизм баз данных, который позволяет осуществить поиск соседних узлов и уточнение их значений длины пути в рамках одного запроса. Для поиска соседних узлов необязательно пробегать весь список ребер, т. к. средства индексирования позволяют сразу обратиться к нужной записи.

```
for (int i = 0; i < N; i++){
   visit [i] = false;
   Answer[i] = Graf[begin_node][i];
}
visit [begin_node] = true;

for (int i = 0; i < N - 1; i++) {
   int k = 0;
   double min_path = numeric_limits<double>::max( );
   // Находим минимальное расстояние до непомеченных вершин
   for (int j = 0; j < N; j++) {
      if (!visit[j] && minras > Answer[j]) {
         min_path = Answer[j];
         k = j;
      }
   }
   // Вершина k помечается просмотренной
   visit [k] = true;

   for (int j = 0; j < N; j++) {
      if (!visit[j] && Answer[j] > Answer[k] + Graf [k][j]) {
         Answer[j] = Answer[k] + Graf [k][j];
      }
   }
}
```

Рис. 1. Листинг алгоритма Дейкстры на языке C++

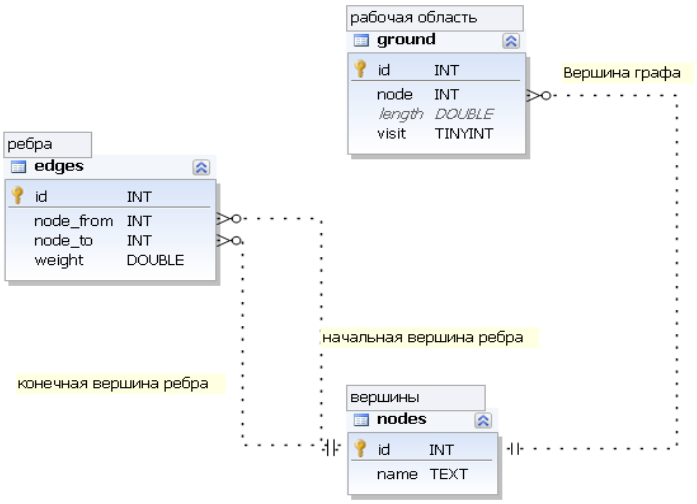

Рис. 2. Структура базы данных

```
CREATE PROCEDURE main(IN _id INT)
  SQL SECURITY INVOKER
BEGIN
  #объявить переменную выхода из курсора, вспомогательные переменные в курсорах
  DECLARE is_end INT DEFAULT 0;
  DECLARE cur_node INT;
  DECLARE step_cnt INT;

  # курсор: еще не покрашенные вершины с определенным расстоянием до них
  DECLARE cur CURSOR FOR
    SELECT
      ground.node
    FROM ground
    WHERE (ground.visit=0)
      AND (NOT ISNULL(ground.length))
    ORDER BY ground.length;
  # обработчик ошибки выхода за границу выборки, используется в курсоре
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET is_end =1;

  # подготовить стартовые данные для рабочей области
  CALL prepare_ground(_id);

  step: LOOP
    OPEN cur;
    # обнулить счетчик,  обновить флаг выхода
    SET step_cnt = 0,is_end = 0;
    # Обойти все вершины из курсора
    wet: LOOP
      FETCH cur INTO cur_node;
      IF is_end THEN
        LEAVE wet;
      END IF;

      # Обработка соседей и  пометка вершины
      CALL check_distance(cur_node);
      # Увеличиваем счетчик
      SET step_cnt = step_cnt + 1;

    END LOOP wet;
    CLOSE cur;
    # Завершить процесс, если не осталось непомеченных вершин
    IF (step_cnt = 0) THEN
      LEAVE step;
    END IF;

  END LOOP step;

  # Вывести ответ
  SELECT
      ground.node
    , ground.length
  FROM ground;
END

CREATE PROCEDURE dijkstra.check_distance(IN _id INT)
  SQL SECURITY INVOKER
BEGIN
  # Длинна пути до узла _id
  DECLARE length DOUBLE;

  # Записать в length известное расстояние до данной вершины
  SELECT  ground.length INTO length
  FROM ground
  WHERE (ground.node = _id);

  UPDATE (
      # Выбор непосещенных узлов, смежных с узлом _id
      SELECT
          DISTINCT edges.node_to as node
        ,  edges.weight
      FROM edges
      WHERE (edges.node_from = _id)
        AND (NOT node_visit(edges.node_to))
  ) as neighbor
    INNER JOIN ground    # Выбранные узлы в рамках рабочей области
      ON (ground.node = neighbor.node)
  # Проверяем, будет ли это минимальный размер пути: если да, то переприсваиваем, иначе оставляем без изменения
  # NULL обозначает в данном алгоритме бесконечность, следовательно, его при сравнении однозначно меняем
  SET ground.length = neighbor.weight + length
  where (ISNULL(ground.length) or (neighbor.weight + length<ground.length));
```

```
  # После просмотра соседей помечаем вершину, говорим, что она посещена
  UPDATE ground
  SET ground.visit = 1
  WHERE (ground.id = _id);

END

CREATE PROCEDURE prepare_ground(IN _id INT)
  SQL SECURITY INVOKER
BEGIN
  #очищаем все строки таблицы
  TRUNCATE TABLE ground;

  #вставляем в таблицу ground  строки из таблицы nodes
  INSERT INTO ground
  SELECT
     NULL as id
    , nodes.id as node
    , IF(nodes.id=_id,0,null) as length
    , 0 as visit
  FROM nodes;
END

CREATE FUNCTION node_visit(_id INT)
  RETURNS tinyint(1)
  SQL SECURITY INVOKER
BEGIN
  DECLARE visit BOOL;

  SELECT ground.visit INTO visit
  FROM ground
  WHERE (ground.node = _id);

  RETURN IFNULL(visit,-1);
END
```

Рис. 3. Листинг хранимых процедур MySQL алгоритма Дейкстры

Теперь рассмотрим аппаратную составляющую и существенное различие в данных подходах. Как уже было сказано, программы, написанные на C++, используют в большей мере ресурсы оперативной памяти, в то время как для работы служб баз данных основанная нагрузка направлена на процессор. В нашем случае для хранения таблиц использовались стандартные средства баз данных, что предполагает работу с памятью жесткого диска и тем самым отрицательно сказывается на скорости. Это заметно при решении задач малой размерности, но с увеличением объема задач алгоритмические преимущества БД-подхода смогли компенсировать временные потери. Данный способ заметно медленнее подхода C++, но при этом позволяет решать намного большие по объему задачи. Как будет видно по результатам тестирования программ, переполнение оперативной памяти большими объемами данных негативно влияет на скорость работы программы, в то время как подход MySQL работает стабильно как с задачами небольшого объема, так и с огромными вычислениями.

Следует отметить, что на тестах более 16000 вершин на данной машине программа на C++ не работала вследствие отсутствия достаточного количества оперативной памяти.

Это можно объяснить тем, что в случае с C++ рабочая матрица весовых коэффициентов имеет размерность n^2 (где n – количество вершин), а значения весовых коэффициентов есть величины действительного типа double (8 байт). Получается, что для задачи в 100 000 вершин потребуется 8 · 10^10 байт, что приблизительно равняется 74,5 Гб. Не всякая оперативная память, даже при поддержке файла подкачки, способна справится с такими объемами. Что касается БД, то здесь вычисления проводятся непосредственно со списком ребер и формула объема памяти такова: $(2 \cdot 4 + 8) \cdot m$, где m – количество ребер графа.

Для сравнения быстродействия алгоритмических подходов был проведен ряд экспериментов по решению задач различной размерности. Для тестирования использовался ПК со следующими системными характеристиками: процессор Intel Celeron CPU 2.66 GHz, оперативная память Kingston memory 988Mb.

В результате экспериментов были получены результаты, которые представлены в табл. 1.

Таблица 1

Результаты тестирования

Количество вершин	Время выполнения (с)	
	MySQL	C++
100	0,3	0
200	0,6	0
500	1,5	0
1000	3,1	0,04
2000	7,4	0,13
5000	28,2	0,75
10000	90,1	40
10500	95	73
11000	112,9	115,2
12000	123,5	182,3
14000	160,2	549,9
15000	172,7	809

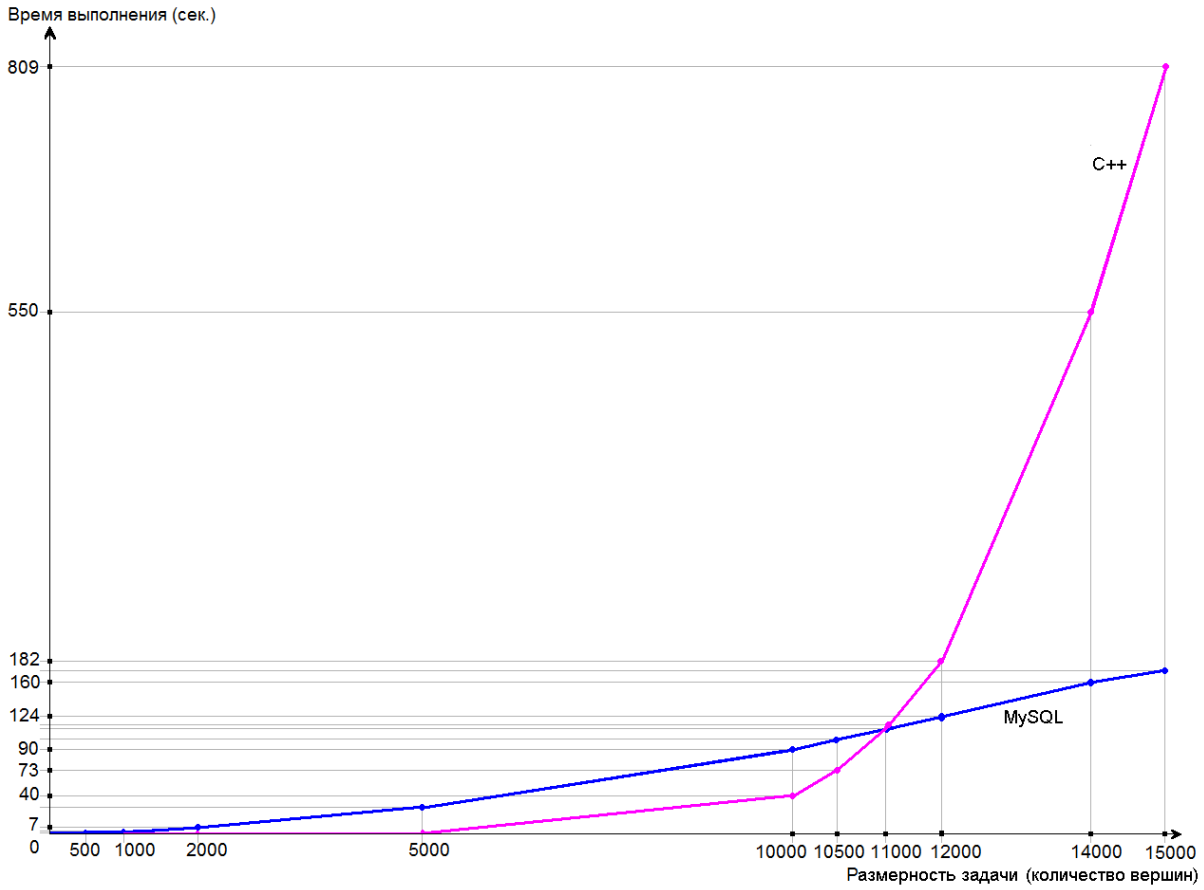

Рис. 4. График результатов тестирования

Для наглядности результаты тестирования проиллюстрированы на графике (рис. 4).

Таким образом, из алгоритмических преимуществ решения данной задачи средствами БД наиболее существенны два момента: уточнение расстояний до непосещенных смежных вершин в рамках единственного запроса UPDATE и ускоренный поиск вершин посредством встроенных в ядро СУБД инструментов индексирования.

Основным же *недостатком данной реализации* является сравнительно низкая скорость выполнения единичных запросов к базе данных, что заметно при решении задач малой и средней размерности.

Направлением дальнейшей деятельности по данному вопросу предполагается исследование способов повышения производительности вычислений, применение методов СУБД в приложениях по математическим расчетам, а также реализация иных математических алгоритмов средствами баз данных и их исследование и оптимизация.

ЛИТЕРАТУРА

1. *Алексеев В.Е.* Графы и алгоритмы. Н. Новгород, 2000. С. 21-23.
2. *Майника Э.* Алгоритмы оптимизации на сетях и графах. М.: Мир, 1981. С. 42-53.
3. *Свами М., Тхуласираман К.* Графы, сети и алгоритмы. М.: Мир, 1984. С. 5-30.

Поступила в редакцию 2 сентября 2012 г.

Markeev V.Y., Kryuchkov A.A. USE OF DATABASES FOR MATHEMATICAL CALCULATIONS ON THE EXAMPLE OF DIJKSTRA'S ALGORITHM

Realization of Dijkstra's algorithm by means of databases is considered. The benefits of this approach for large-scale problems are shown.

Key words: graph theory; problem on minimal path; Dijkstra's algorithm; databases; stored procedures.