Алгоритмы поиска путей на графах большого размера

И. В. ПОЛЯКОВ
Национальный исследовательский университет «Высшая школа экономики»
e-mail: igorp86@mail.ru

А. А. ЧЕПОВСКИЙ
Национальный исследовательский университет «Высшая школа экономики»
e-mail: c4hapa@gmail.com

А. М. ЧЕПОВСКИЙ
Национальный исследовательский университет «Высшая школа экономики»
e-mail: achepovskiy@hse.ru

УДК 004.421.2:519.178

Ключевые слова: поиск путей в графе, алгоритм Дейкстры, метод спектральной бисекции, метод k-средних.

Аннотация

В статье исследованы различные подходы к хранению и обработке данных. Предложен новый алгоритм поиска путей между вершинами графа больших размеров.

Abstract

I. V. Polyakov, A. A. Chepovskiy, A. M. Chepovskiy, Algorithms for searching paths in huge graphs, Fundamentalnaya i prikladnaya matematika, vol. 19 (2014), no. 1, pp. 165—172.

Various approaches for data storing and processing are investigated in this paper. A new algorithm for finding paths in a huge graph is introduced.

1. Общая характеристика проблематики

Задача поиска кратчайшего пути является одним из самых востребованных на практике аспектов современной теории графов. Данная задача может быть поставлена в нескольких различных вариантах. В случае графа без каких-либо дополнительных структур данных наиболее известным и широко используемым является классический алгоритм Дейкстры [2]. В худшем случае его трудоёмкость оценивается величиной $O(m + n \cdot \log n)$ [4], где n — число вершин графа, а m — число его рёбер. В случае неориентированного графа обычно применя-

Фундаментальная и прикладная математика, 2014, том 19, № 1, с. 165—172.

ется двунаправленный поиск Дейкстры. В этом случае поиск запускается из двух вершин, между которыми ищется путь. Если веса всех рёбер одинаковы, кратчайший путь может быть найден поиском в ширину за $O(m+n)$ операций. В общем случае не известны алгоритмы, улучшающие данную оценку. Тем не менее для некоторых классов графов она может быть улучшена. Так, в [3] представлен алгоритм для планарных графов, требующий практически линейное количество операций для предобработки. Данный алгоритм позволяет находить кратчайшие пути с трудоёмкостью, практически линейно зависящей от длины этого пути, пока размер пути не превысит определённого порогового значения.

На практике для поиска путей в больших графах требуется дополнительная предобработка, создающая набор эвристик, позволяющих впоследствии оптимизировать работу алгоритма Дейкстры.

Среди предлагаемых подходов можно отметить предложенные в [5] и [6]. Все предлагаемые эвристики ориентированы на случай фиксированного графа, полностью помещающегося в оперативной памяти. При этом они могут быть вычислены за разумное время на графах, содержащих порядка нескольких миллионов рёбер и вершин. В случае графов большего размера они, как правило, оказываются неприменимыми.

В данной работе предложен подход к хранению больших графов, ориентированный на поиск путей в постоянно модифицируемых графах большого размера. При этом предполагается, что размеры обрабатываемых графов столь велики, что не позволяют хранить их в оперативной памяти целиком. Кроме того, предполагается, что в запросе на поиск пути могут быть дополнительные предикаты, задающие какие-либо условия на рёбра, которые должны войти в найденный путь. Используемые эвристики позволяют значительно оптимизировать количество требуемых дисковых операций по сравнению с классическим алгоритмом Дейкстры. Основная идея предложенного решения состоит в разбиении графа на относительно небольшие кластеры K_i, $i = 1, \ldots, L$, с высокой плотностью связей. Размеры данных кластеров должны позволять разместить каждый из них в оперативной памяти. Процесс разбиения графа на кластеры производится таким образом, чтобы плотность связей в них превышала среднюю плотность связей в графе. Каждый кластер занимает непрерывную область дисковой памяти, следовательно, он может быть загружен в память без дополнительных операций позиционирования головки диска.

Двусторонний поиск Дейкстры производится до тех пор, пока из двух вершин не будет достигнут один и тот же кластер. В этом случае кластер загружается в память и путь ищется внутри него. Кроме того, при выборе вершин, просматриваемых на очередном этапе, предпочтение отдаётся тем вершинам, которые находятся в наиболее плотных кластерах из тех, которые ещё не были просмотрены.

2. Основные определения и обозначения

Обозначим множество всех узлов графа через $V = \{v_1, \ldots, v_n\}$, а множество рёбер — через $E = \{e_1, \ldots, e_m\}$. Предполагается, что в графе могут быть кратные рёбра с некоторым количеством атрибутов. Атрибуты вершины v будем обозначать через $A_i(v)$, $i = 1, \ldots, N$, а атрибуты ребра e — через $B_i(e)$, $i = 1, \ldots, M$. Предполагается, что каждая из функций A_i, B_i отображает множество всех возможных вершин или рёбер в пространство значений X_i соответствующего атрибута. Все такие пространства должны содержать специальный элемент NULL. Предполагается, что $A_i(v) = \mathrm{NULL}$, если для данной вершины i-й атрибут не определён. Аналогичное соглашение выполняется для рёбер. На каждом пространстве X_i определена некоторая хэш-функция H_i, отображающая элементы данного пространства во множество натуральных чисел. Для каждой вершины v под $\mathrm{E}(v)$ будем понимать все рёбра, инцидентные данной вершине. Для каждого ребра e обозначим через $\mathrm{V}(e) = \big(v_1(e), v_2(e)\big)$ множество инцидентных ему вершин.

3. Особенности хранения графа

Граф представлен двумя дублирующими друг друга структурами данных. Первая структура хранит списки смежностей всех вершин и постоянно пополняется. Поскольку количество вершин графа потенциально может быть очень велико, все списки смежностей соединяются в фиксированное количество списков (порядка 100 000). Предположим, что каждая вершина v адресуется некоторым натуральным числом $\mathrm{ind}(v)$. Тогда можно хранить все связи вершины v в списке с номером $\mathrm{ind}(v) \pmod{100\,000}$. Для того чтобы получить все связи вершины v, необходимо целиком прочитать список с номером $\mathrm{ind}(v) \pmod{100\,000}$, отфильтровав его соответствующим образом. Каждый из 100 000 списков представляет собой двунаправленный список дисковых блоков фиксированного размера (порядка 400 Кб). При этом все списки снабжены буфером в оперативной памяти (порядка 40 Кб) для амортизации операций добавления новых связей. Отметим, что если количество кластеров относительно невелико (меньше 100 000), становится возможным назначить вершинам каждого кластера отдельный список. Тогда в оперативной памяти дополнительно должен храниться массив, ставящий в соответствие каждому натуральному числу $\mathrm{ind}(v)$ номер списка, соответствующего кластеру вершины v.

Вторая структура данных хранит в себе всю информацию, связанную с кластерами вершин. Каждый кластер занимает непрерывную область дисковой памяти. В ней содержатся все внутрикластерные связи графа, а также произвольные дополнительные структуры данных, позволяющие оптимизировать поиск путей в кластере. Примеры таких структур описаны в [5, 6].

Также в оперативной памяти присутствует структура, обеспечивающая отображение номеров вершин в соответствующие им кластеры. Это может быть

обычный массив. Если количество кластеров на каком-то этапе алгоритма кластеризации позволит хранить в оперативной памяти граф связей между кластерами, в оперативной памяти должна быть создана дополнительная структура, отвечающая этому графу.

4. Алгоритм кластеризации вершин

Первоначальное разбиение на кластеры производится случайным образом. После каждого проведённого процесса кластеризации все полученные кластеры следует разбить на компоненты связности, что может быть сделано за $O(m+n)$ операций. Изолированные вершины очевидным образом не могут участвовать в построении путей. Для их хранения создаётся специальный кластер изолированных вершин. По мере накопления связей запускается процесс перестроения кластеров методом k-средних [7]. Он производится итерационным образом. На каждой итерации для каждой вершины v и каждого кластера K_j вычисляется доля $\rho(v, K_j)$ связей данной вершины, ведущих в данный кластер:

$$\rho(v, K_j) = \frac{|\{e \in \mathrm{E}(v)\colon v_1(e) \in K_j \text{ или } v_2(e) \in K_j\}|}{|\mathrm{E}(v)|}.$$

Затем вершина перемещается в кластер, для которого эта характеристика получилась максимальной. Количество операций, требуемых на одном шаге, составляет $O(|E|)$. Итерационный процесс проводится несколько раз. Критерием остановки является условие $C_j/|V| < \varepsilon$, где C_j — количество вершин, поменявших кластер на j-й итерации, а ε — параметр алгоритма. Если из-за слишком большого размера невозможна загрузка кластеров в оперативную память, производится их разделение. Для этого используется метод спектральной бисекции, предложенный в [1]. Бисекцию следует производить в момент очередного прохода по методу k-средних, когда размер одного из аккумулируемых кластеров становится больше половины критического размера, при превышении которого загрузка кластера в оперативную память становится невозможной. После произведённого разбиения кластера на два подкластера очередная отнесённая к нему вершина, относится к одному из полученных подкластеров по количеству связей с каждым из них.

В данном разделе через n обозначается число вершин не в исходном графе, а в разбиваемом кластере K. Для K строится матрица Лапласа (Кирхгофа) $L = (l_{ij})$, недиагональные элементы которой совпадают с элементами матрицы смежностей графа, а на диагонали стоят степени соответствующих вершин со знаком минус:

$$l_{ii} = -\deg v_i,$$

$$\text{при } \ i \neq j \quad l_{ij} = \begin{cases} 1, & \text{если } v_i \text{ смежна с } v_j, \\ 0 & \text{иначе.} \end{cases}$$

Любое разбиение K на два кластера K' и K'' может быть описано с помощью вектора $\bar{s} = (s_1, \ldots, s_n)$:

$$s_i = \begin{cases} 1, & \text{если вершина } v_i \text{ оказалась в кластере } K', \\ -1, & \text{если вершина } v_i \text{ оказалась в кластере } K''. \end{cases}$$

При этом известно, что величина $C(s) = (1/4)s^T Ls$ в точности равна количеству рёбер между кластерами K' и K'', задаваемыми вектором $\bar{s}$. С другой стороны, симметричный оператор L имеет базис из собственных векторов $\bar{e}_1, \ldots, \bar{e}_n$, отвечающих собственным значениям $0 \leqslant \lambda_1 \leqslant \ldots \leqslant \lambda_n$. Если разложить вектор s по этому базису,

$$\bar{s} = \sum_{i=1}^{n} \mu_i \bar{e}_i,$$

то значение $C(s)$ выражается как

$$\frac{1}{4} \sum_i \lambda_i (\mu_i)^2.$$

Соответственно, для минимизации данной величины выгодно в качестве вектора $\bar{s}$ выбрать вектор, для которого коэффициенты разложения по собственным векторам, отвечающим минимальным собственным значениям, максимальны. В классическом методе бисекции выбирается первый собственный вектор $(x_1, \ldots, x_n)$, отличный от тривиального $(1, 1, \ldots, 1)$, отвечающего собственному значению 0. Вектор $\bar{s}$ строится по вектору $\bar{x}$ следующим образом:

$$s_i = \begin{cases} 1, & \text{если } x_i \geqslant 0, \\ -1 & \text{иначе.} \end{cases}$$

Так максимизируется скалярное произведение с вектором $\bar{x}$. В используемом в данной статье методе такой подход модернизирован, поскольку требуется получить разбиение на два кластера одинакового размера. Для координат вектора $\bar{x}$ выбирается медианное значение M. Вектор $\bar{s}$ строится по вектору $\bar{x}$ следующим образом:

$$s_i = \begin{cases} 1, & \text{если } x_i \geqslant M, \\ -1 & \text{иначе.} \end{cases}$$

Отметим, что в данном методе в качестве вектора $\bar{x}$ можно брать сумму нескольких собственных векторов, отвечающих нескольким минимальным собственным значениям. Для поиска собственных векторов используется итерационный метод. Собственный вектор, отвечающий максимальному собственному значению неотрицательной симметричной матрицы, может быть найден как предел последовательности

$$\left\{ \frac{L^n a}{|L^n a|} \right\},$$

где $\bar{a}$ — некоторый случайно выбранный вектор. Следующий по старшинству вектор ищется как предел

$$\left\{ \frac{L^n b}{|L^n b|} \right\},$$

где $\bar{b}$ — случайно выбранный вектор, ортогональный собственному вектору, отвечающему максимальному собственному значению. Для повышения точности вычислений нормировку и ортогонализацию получаемого вектора следует проводить на каждой итерации. Для минимизации значения $C(s)$ требуется искать минимальные собственные вектора, следовательно, необходимо рассмотреть матрицу $d \cdot E - L$, где E — единичная матрица, d — константа, превышающая максимальное собственное значение матрицы L. Отметим, что для каждой итерации данного алгоритма требуется $O(m+n)$ операций, поскольку в матрице L имеется $O(m+n)$ ненулевых элементов. При этом саму матрицу L строить и хранить необязательно, достаточно сохранить лишь её ненулевые элементы, т. е. иметь представление графа в виде списков смежности. Описанный процесс кластеризации запускается после добавления очередной порции вершин и связей. В случае непрерывного добавления целесообразно выполнять его после каждого удвоения числа связей в графе.

5. Поиск путей с использованием кластеров

Рассмотрим задачу нахождения какого-либо пути между множествами вершин S_1 и S_2 за минимальное время. При этом желательно найти путь как можно меньшей длины. Обозначим через V_i множество вершин, ассоциированных с кластером K_i. Если в оперативной памяти хранится граф связей кластеров, то кратчайший путь сначала ищется в нём с помощью обычного поиска в ширину (возможно использование дополнительных эвристических характеристик, описанных в [5,6]). Если путь $(K_1, \ldots, K_l)$ между кластерами успешно найден, его можно конкретизировать за линейное время от суммарного размера кластеров. Для этого в оперативную память последовательно загружаются все связи каждого из кластеров, после чего находятся вершины $(ve_1, vb_2, ve_2, \ldots, ve_L)$, такие что ve_i смежна с vb_{i+1}, при этом $vb_i, ve_i \in K_i$. Далее необходимо последовательно загрузить все связи каждого из кластеров, найдя все пути между vb_i и ve_i. Искомый путь будет композицией найденных путей. Если путь между кластерами множеств S_1 и S_2 отсутствует, необходимо использовать следующий алгоритм.

На каждом шаге предлагаемого алгоритма образуются множества S_1^n, S_2^n, а также некоторые структуры данных D_1^n, D_2^n, ставящие в соответствие каждой вершине $s \in S_j^n$ некоторую вершину-предок $D_j^n(s)$. В качестве такого рода структур могут быть использованы обычные массивы. При этом $S_j^1 = S_j$ и $D_j^1(s) = s$. Опишем n-й шаг алгоритма.

1. Если множества S_1^n, S_2^n пересекаются, то кратчайший путь найден. Для его вычисления используются структуры D_1^n, D_2^n.

2. Проверяем существование таких i, для которых $S_1^n \cap V_i$ и $S_2^n \cap V_i$ одновременно не пусты.

3. Для каждого найденного i загружаем i-й кластер в память, строим по нему граф G_i и ищем в нём кратчайшие пути между множествами $S_1^n \cap V_i$, $S_2^n \cap V_i$. Каждый такой путь очевидным образом даёт некоторый путь между множествами S_1^n, S_2^n в исходном графе, который с помощью структур D_1^n, D_2^n продолжается до некоторого пути между S_1 и S_2. Если такие пути существуют, то добавляем их в список найденных путей. Полученные таким образом пути не обязательно являются кратчайшими, поэтому работа алгоритма может быть продолжена с целью поиска новых более коротких путей.

4. Добавляем в множество S_1^{n+1} все вершины из множества S_1^n, а также все смежные с ними вершины. Для каждой вершины, не содержащейся в S_1^n, запоминаем ту вершину, из которой она была достигнута в структуре D_1^{n+1}. Таким же образом строим множество S_2^{n+1}, после чего снова переходим к шагу 1. Шаг 4 может быть оптимизирован с использованием следующих соображений:

а) расширение множеств S_1^n, S_2^n выгодно проводить одновременно, сразу проверяя выполнение условий из шагов 1 и 2. Для этого следует последовательно поочередно производить загрузку списков смежности вершин из S_1^n, S_2^n;

б) обычный поиск в ширину может быть заменён A^*-поиском [5]. Для расширения множеств S_1^n, S_2^n можно загружать списки смежности тех вершин, кластеры которых наименее полно представлены в множествах S_1^n, S_2^n. Если в оперативной памяти присутствует граф всех кластеров, в процессе его предобработки можно вычислить центральность каждого кластера — количество кратчайших путей между всеми парами кластеров, проходящих через него, после чего в процессе для расширения множеств S_1^n, S_2^n следует выбирать вершины, принадлежащие кластерам с наибольшей центральностью. В качестве целевой функции A^*-поиска можно выбрать

$$f(v) = \frac{\mathrm{C}\big(K(v)\big)}{d(v, S_j)},$$

где $d(v, S_j)$ — расстояние от вершины v до соответствующего множества S_j, а $\mathrm{C}\big(K(v)\big)$ — центральность кластера вершины v.

6. Заключение

Предложенный алгоритм предобработки и хранения графов большого размера позволяет оптимизировать поиск пути между множествами вершин. Используемый для поиска алгоритм кластеризации является итерационным с ли-

нейным числом операций, требуемым для проведения одной итерации. Используемая в поиске пути оценочная функция выбирает вершины, которые могут с высокой вероятностью привести двусторонний поиск в один кластер, после чего оставшаяся часть пути ищется только в данном кластере. В результате предложенного алгоритма кластеризации получаемые кластеры оказываются связными, что гарантирует наличие в них некоторого пути. Если поиск производится с дополнительными условиями на атрибуты, попадание в один кластер не гарантирует наличия пути. Однако вероятность существования пути в кластере, удовлетворяющего заданным предикатам, всё же довольно высока из-за высокой плотности связей в кластере. Данный подход может быть скомбинирован с подходами, предложенными в [5, 6]. Для этого следует рассчитать предлагаемые эвристики для каждого из кластеров графа. Это позволит оптимизировать поиск пути без дополнительных предикатов внутри одного кластера графа.

Литература

[1] Barnes E. R. An algorithm for partitioning the nodes of a graph // SIAM J. Algebraic Discrete Methods. — 1982. — Vol. 4, no. 3. — P. 541—550.

[2] Dijkstra E. W. A note on two problems in connexion with graphs // Numer. Math. — 1959. — Vol. 1. — P. 269—271.

[3] Fakcharoenphol J., Rao S. Planar graphs, negative weight edges, shortest paths, and near linear time // Proc. 42nd IEEE Symp. Foundations of Computer Science. — 2001. — P. 232—241.

[4] Fredman M. L., Tarjan R. E. Fibonacci heaps and their uses in improved network optimization algorithms // J. ACM. — 1987. — Vol. 34, no. 3. — P. 596—615.

[5] Goldberg A. V., Harrelson C. Computing the shortest path: A^*-search meets graph theory // Proc. Sixteenth Annual ACM—SIAM Symp. on Discrete Algorithms, January 23—25, Vancouver, BC (2005). — P. 156—165.

[6] Hilger M., Köhler E., Möhring R. H., Schilling H. Fast point-to-point shortest path computations with arc-flags // The Shortest Path Problem: Ninth DIMACS Implementation Challenge C. Demetrescu, A. V. Goldberg, D. S. Johnson, eds. — Amer. Math. Soc., 2009. — (DIMACS Book; Vol. 74). — P. 41—72.

[7] Steinhaus H. Sur la division des corps matériels en parties // Bull. Acad. Polon. Sci. Cl. III. — 1956. — No. 4. — P. 801—804.