Санкт-Петербургский национальный исследовательский университет
информационных технологий, механики и оптики

Кафедра компьютерных технологий

Г. С. Ткаченко

Параллельные алгоритмы поиска кратчайшего пути в графе

Бакалаврская работа

Научный руководитель: Г. А. Корнеев

Санкт-Петербург
2015

Содержание

Введение

Алгоритмы поиска кратчайших путей на графах нашли свое применение в различных областях и сферах деятельности человека. Такие алгоритмы используются в картографических сервисах, при построении пути GPS-навигатора, для представления и анализа дорожной сети и во многих других областях.

При этом в настоящее время существует большое число алгоритмов и подходов, которые решают эту задачу. И все алгоритмы можно логически разделить на два класса - алгоритмы поиска кратчайшего расстояния от одной вершины до всех остальных и алгоритмы поиска кратчайших расстоянии между каждой парой вершин. Из первого класса самыми яркими представителями являются различные модификации алгоритмов Дейкстры и Беллмана-Форда. Для решения задач второго класса часто используются алгоритмы Флойда-Уоршелла и алгоритм Джонсона.

Однако с ростом многопроцессорных архитектур встала задача по возможности запускать эти алгоритмы на нескольких вычислительных ядрах. Именно такие параллельные версии алгоритмов будут освещены в моей работе. В первом части представлены параллельные модификации алгоритма Беллмана-Форда. Во второй части работы представлен алгоритм по поиску кратчайших расстояний между каждой парой вершин в общем случае и эффективная модификация для поиска расстояний в социальных графах.

Глава 1. Решение задачи поиска кратчайшего расстояния от фиксированной вершины до всех остальных

В данной главе описаны алгоритмы по решению классической задаче на графах - поиску кратчайших расстояний от одной вершины до всех остальных. В первой части главы представлен краткий обзор предметной области. Во второй параллельные модификации алгоритма Беллмана-Форда.

1.1. Обзор существующих решений

1.1.1. Алгоритм Дейкстры

Одним из наиболее заметных алгоритмов для решения данной задачи является алгоритм Дейкстры. Придуманный еще в 1959 году Эдсгером Вибе Дейкстрой он сохраняет свою актуальность и по сей день. Основная идея состоит в последовательном пополнении множества вершин, расстояние для которых уже корректно посчитано. При этом на каждом шаге выбирается вершина, которая находится ближе остальных к уже посчитанному множеству.

Существует множество модификации алгоритма основанных на различных структурах данных для выбора вершины с минимальный расстоянием на каждом из шагов алгоритма. В зависимости от этого алгоритм может работать O(V*V+E), O(ElogV) или O(VlogV+E).

Основная проблема алгоритма состоит в том, что он работает только на графах с неотрицательным весом ребер. С этой проблемой справляется алгоритм Беллмана-Форда.

1.1.2. Алгоритм Беллмана-Форда

Классический алгоритм Беллмана-Форда работает на графах с произвольным весом ребер, однако имеет заметно худшую асимптотику по сравнению с алгоритмом Дейсктры - O(VE).

Основная идея алгоритма основана на идее динамического программирования. После k итерации алгоритма утверждается, что будут корректно посчитаны и обработаны значения веса путей длиной не более K. И после V итерации расстояние до каждой из вершин посчитано корректно. Ниже приведен каноничный псевдокод алгоритма.

Алгоритм 1 Классический алгоритм Беллмана-Форда

procedure CLASSICBELLMANFORD($G, start$)
  $dist \leftarrow \{\infty ... \infty\}$
  $dist[start] \leftarrow 0$
  for $i = 0$ to $|G.vertices|$ **do**
    for $e \in G.edges$ **do**
      $dist[e.to] \leftarrow \min(dist[e.to], dist[e.from] + e.w)$
  return $dist$

Кроме того существует интересная модификация алгоритма, которая поддерживает на каждой итерации набор вершин, расстояние до которых изменились на предыдущем шаге алгоритма. Из очевидных соображений мы имеем право рассматривать только эти и никакие другие вершины. Этот алгоритм на практике зачастую работает заметно быстрее чем классическая версия в некоторых случаях, но об этом подробно будет описано позднее. Будем называть эту версию BFS-подобный Беллман-Форд. Ниже приведен псевдокод алгоритма

Алгоритм 2 BFS-подобный Беллман-Форд

```
procedure BFSBellmanFord(G, start)
    dist ← {∞...∞}
    dist[start] ← 0
    VertexSet ← {start}                          ▷ структура данных для хранения набора вершин
    NextVertexSet ← ∅
    step ← 0
    while step < |G.vertices| and not VertexSet.empty() do
        step + +
        NextVertexSet.clear()
        for v ∈ VertexSet do
            for e ∈ G.edgesFrom[v] do            ▷ исходящие ребра из текущей вершины
                if dist[e.to] < dist[e.from] + e.w then
                    dist[e.to] ← dist[e.from] + e.w
                    NextVertexSet.insert(e.to)
        VertexSet ← NextVertexSet
```

1.1.3. Другие алгоритмы

Также известны специализированные алгоритмы, такие как алгоритм A* и D*, которые оперирует большими специализированными графами и используют ряд эвристик для поиска расстояний. При этом в контексте наших исследований они затронуты не будут.

1.2. Параллельный алгоритм Беллмана-Форда

В предыдущей главе были рассмотрены классические алгоритмы поиска кратчайших путей в графе. В этой главе будет рассмотрен алгоритм Беллмана-Форда в контексте параллельных вычислений. Кроме того будем использовать в каждом из алгоритмов идею ранней остановки - если на текущем шаге ни одно из значений массива расстояний не изменилось, то имеем право выйти из основного цикла. В последующих главах будет представлено несколько версий алгоритма, а также их последующее сравнение и рекомендации по использованию.

1.2.1. Параллелизация по ребрам вершины

Первая версия алгоритма основана на параллельной обработке всех ребер, исходящих из текущей вершины. Псевдокод, который практически не отличается от классической версии, приведен ниже.

Алгоритм 3 Параллельный Беллман-Форд по ребрам вершины

```
procedure BellmanFordPar1(G, start)
    dist ← {∞...∞}
    dist[start] ← 0
    for i = 0 to |G.vertices| do
        changed ← false
        for v ∈ G.vertices do
            parfor e ∈ G.fromEdges[v] do
                if dist[e.to] < dist[e.from] + e.w then
                    dist[e.to] ← dist[e.from] + e.w
                    changed ← true
                dist[e.to] ← min(dist[e.to], dist[e.from] + e.w)
        if not changed then
            break
    return dist
```

1.2.2. Параллелизация по всем ребрам

Идея второго алгоритма состоит в разбиений всего набора вершин на некоторые подмножества, каждое из которых будет обрабатываться отдельным процессором. При этом для каждой вершины будем рассматривать набор ребер, входящих в нее. Это необходимо для того, чтобы обновление фиксированной ячейки в массиве расстояний происходило только одним потоком.

Алгоритм 4 Параллельный Беллман-Форд по всем ребрам

```
procedure BellmanFordPar2(G, start)
    dist ← {∞...∞}
    dist[start] ← 0
    prefsum ← prefix sum of vertices incoming degree
    planMap ← BuildPlan(prefsum, 0, |G.vertices|
    for i = 0 to |G.vertices| do
        if not ProcessLayer(G, planMap, prefsum, 0, |G.vertices|) then
            break
    return dist

procedure BuildPlan(prefsum, startV, endV)    ▷ Функция возвращают структуру, которая по отрезку возвращает его середину по количеству ребер
    edgesNumber ← prefsum[endV] − prefsum[startV]
    if edgesNumber < threshold then
        midV ← binary search on edges number
        resultMap[startV][endV] ← midV
        resultMap[startV][endV].insert(BuildPlan(prefsum, startV, midV)
        resultMap[startV][endV].insert(BuildPlan(prefsum, midV, endV)
    return resultMap

procedure ProcessLayer(G, planMap, prefsum, startV, endV)
    edgesNumber ← prefsum[endV] − prefsum[startV]
    if edgesNumber < threshold then
        process vertices sequentally
    else
        midV ← planMap[startV][endV]
        ProcessLayer(G, planMap, prefsum, startV, midV)
        ProcessLayer(G, planMap, prefsum, midV, endV)
```

1.2.3. Параллелизация BFS - версии

Предыдущие две версии были основаны на параллелизации классической версии Беллмана-Форда. В основе следующего алгоритма лежит идея обхода в ширину (Алгоритм 2). В качестве основы для параллельной версии такого алгоритма был взят параллельный обход в ширину, предложенный Умутом Акаром и Майком Рэйни. Подробнее о внутреннем устройстве их подхода речь зайдет во 2 части работы в контексте решения задачи поиска расстояний между каждой парой вершин. Пока лишь приведем псевдокод

Алгоритм 5 Параллельный BFS-подобный Беллман-Форд

```
procedure BellmanFordPar3(G, start)
    dist ← {∞...∞}
    dist[start] ← 0
    Frontier ← {start}   ▷ структура данных для хранения исходящих ребер текущего множества
    for i = 0 to |G.vertices| do
        Frontier ← handleFrontier(Frontier)   ▷ релаксируем ребра из фронтира и строим новый
        if Frontier.empty() then
            break
    return dist

procedure HandleFrontier(Frontier)
    recursively divide current frontier, atomically relax edges in frontier and building a new one
    return NewFrontier
```

1.2.4. Сравнение подходов

Каждый из вышеизложенных подходов имеет свои особенности, что позволяет каждому из них конкурировать друг с другом на некоторых типах графов. Рассмотрим эти особенности.

С первого взгляда может показаться, что Алгоритм 3 имеет лишь одни недостатки - он имеет наименьшим образом по сравнению с последующими задействует все процессоры и при этом асимптотически равен остальным двум. Однако рассмотрим внимательнее каноничный аглоритм Беллмана-Форда и запустим его на плотном графе, где для каждого ребра верно, что индекс вершины источника меньше индекса вершины назначения. В этом случае каноничной версии достаточно будет сделать лишь две итерации внешнего цикла, поскольку на каждой итераций внутреннего цикла значение расстояния для текущей вершины будет корректно посчитано (очевидно доказывается по математической индукции). Так как количество итерации третьего алгоритма в подобных графах может быть значительным и размер текущей очереди может быть большим, а второму же алгоритму на таких графах неплохая способность параллелиться будет только вредить - она будет заметно увеличивать число итераций внешнего цикла. То есть в таких случаях последние два алгоритма работают хуже первого.

Но очевидно, что в большинстве случаев последние два алгоритма будут показывать лучшие результаты. Сравним эти два подхода. Алгоритм, основанный на обходе в ширину, заметно сокращает количество вершин для обработки в пределах каждой итерации. Однако на плотных графах количество таких вершин значительно и такой подход показывает себя не с лучшей стороны.

1.2.5. Тестирование

Для подтверждения вышеприведенных замечаний все вышеизложенные подходы были реализованы на основе библиотеки для параллельных вычислений PASL. Тестирование производилось на ряде графов на машине 40-core Intel machine (with hyper-threading) with 4×2.4GHz Intel 10-core E7-8870 Xeon processors, a 1066MHz bus, and 256GB of main memory.

Алгоритмы тестировались на различных графовых структурах, которые перечислены в Таблице 1.1. При этом описание для формата приведено внизу таблицы.

Name	Vertices	Edges
Complete TS	7071	24995985
Complete sign	3162	9995082
BalancedTree fraction	8388607	8388608
SquareGrid sign	2499561	4999122
RandomSparse fraction(0.5) sign	2500000	25000000
RandomSparse fraction(0.96) sign(+)	2500000	25000000
RandomDense fraction(0.5) sign	5000	25000000
RandomDense fraction(0.96) sign(+)	5000	25000000

sign - sign of weights on edges

fraction - fraction of lexicographically sorted edges (edges of type X -> X+i)

TS - exists only Lexicographically Sorted edges (fraction = 1)

expression *RandomDense fraction(0.5) sign* means Random Dense graph with specified fraction and any sign of weights

Таблица 1.1: Input graph description

Algo №	Complete			BalancedTree		SquareGrid		RandomSparse			RandomDense		
	TS	+	-	0.5	1	+	+-	0.5+	0.5-	0.96+	0.5+	0.5-	0.96+
3	2.43	4.65	nc	116.31	9.04	5.49	13.40	nc	nc	24.35	nc	nc	5.01
4	5.17	0.18	10.84	3.59	3.08	5.92	7.10	2.77	14.68	2.42	0.48	6.38	0.46
5	44.63	0.37	23.55	0.44	0.31	4.42	0.58	0.98	22.59	0.76	0.60	10.25	0.71

Таблица 1.2: Bellman-Ford algorithms comparison

1.3. Выводы

Из таблиц видно, что наши ожидания относительно применимости конкретных подходов оправдались. Первый из алгоритмов работает лучше на узком спектре графов с высоким fraction и высокой средней степенью вершины, второй хорошо работает на плотных графах и графах с отрицательным весом ребер, третий же заметно лучше остальных на разреженных графах.

Таким образом сделав простой анализ структуры графа мы сможем выбрать необходимый алгоритм для поиска кратчайшего пути.

Глава 2. Решение задачи поиска расстояний между каждой парой вершин графа

В этой главе будет приведено решение задачи поиска расстояний между каждой парой вершин. В начале главы будет краткий обзор предметной области, после будут приведены два наивных решения для поиска пути, а после комбинацией будет приведено решение задачи для социальных неориентированных невзвешенных графов, которое будет сочетать несколько подходов и идей, изложенных в предыдущих алгоритмах.

2.1. Обзор существующих решений

2.1.1. Алгоритм Флойда

Одним из наиболее известных алгоритмов, который применяется для решения данной задачи является алгоритм Флойда. Этот алгоритм использует идею динамического программирования и выполняется за O(VVV). Основная идея состоит в обновлений пути между двумя текущими вершинами выбором некоторой вершины, через который может пройти потенциальный кратчайший путь. Псевдокод алгоритма приведен ниже.

Алгоритм 6 Алгоритм Флойда

```
procedure FLOYD(G)
    dist ← {{∞...∞} ... {∞...∞}}
    for e ∈ G.edges do
        dist[e.from][e.to] ← e.w

    for i = 0 to |G.vertices| do
        for u = 0 to |G.vertices| do
            for v = 0 to |G.vertices| do
                dist[u][v] ← min(dist[u][v], dist[u][i] + dist[i][v])
    return dist
```

2.1.2. Альтернативы

В некоторых случаях оказываются эффективны другие подходы. Например, можно для каждой вершины по отдельности запустить некоторых алгоритм поиска кратчайшего пути до всех остальных вершин. Для случая неотрицательных ребер можно применить алгоритм Дейкстры, в более общем случае может быть применен Беллман-Форд. Кроме вышеприведенных подходов также известен алгоритм Джонсона, который работает на графах без циклов отрицательного веса и находит кратчайшие расстояния за время O(VV * log(V) + VE). Все эти подходы оказываются эффективны в случае разреженных графов.

В последующих подходах в качестве основы для параллельного алгоритма будет использоваться именно идея подсчета расстояний либо для каждой вершины по отдельности, либо подсчета расстояний для групп вершин. И все нижеперечисленные алгоритмы, как и описанные выше альтернативы хорошо работает на разреженных графах.

2.2. Наивная параллельная версия

Первая версия заключается исключительно в запуске Беллмана-Форда для каждой из вершин. При этом заметим, что так как каждый из них независим друг от друга, то можем эти запуски распараллелить между собой. Таким образом, псевдокод из себя представляет следующее

Алгоритм 7 Наивная параллельная версия

```
procedure AllPairsPar1(G)
    return HandleVertices(G, 0, |G.vertices|)

procedure HandleVertices(G, startVertex, endVertex)
    if endVertex − startVertex < threshold then
        run parallel Bellman-Ford for every vertex
        return distances
    else
        midV ← (startVertex + endVertex)/2
        HandleVertices(G, startV, midV)
        HandleVertices(G, midV, endV)
```

2.3. Параллельный алгоритм для объединенного графа

Развитием предыдущей идеи является наблюдение, что для некоторого набора вершин можем построить общий граф и запустить на нем Беллмана-Форда, что потенциально может повысить производительность за счет высокой параллельности каждого отдельно взятого Беллмана-Форда. Кроме того, это избавит нас от выбора константы для предыдущей версии, что упростит использование алгоритма для пользователя.

Идея заключается в запуске алгоритма Беллман-Форд на графе, вершины которого описываются двумя значениями - текущей вершины в обходе и вершины, из которой этот обход начался (иными словами, вершины, из которой мы ищем кратчайшие расстояния). После построения графа будет достаточно запустить обход, при этом положив в Frontier все вершины вида (i, i). В итоге кратчайшее расстояние для вершины (i, j) будет интерпретироваться как кратчайшее расстояние от вершины i до вершины j в исходном графе.

Однако, как будет показано позднее, такой подход на практике оказался медленнее наивной версий. Но при этом идея обработки ряда вершин одновременна легла в основе следующего алгоритма для социальных графов.

2.4. Параллельный алгоритм для социальных графов

В данной главе будет рассмотрен алгоритм поиска кратчайшего пути между каждой парой вершин для графов реальных социальных сетей. При этом рассмотренный граф будет невзвешенный и неориентированный. Кроме того, в этой главе будет затронута структура данных, уже ранее используемая в параллельном обходе в ширину. Будет описан ее интерфейс и принцип работы.

2.4.1. Идея алгоритма

В графах для социальных сетей известна одна эвристика, которая называется "Теория шести рукопожатий". В ее основе лежит тот факт, что практически любые два человека на земле знакомы не более, чем через пятерых промежуточных людей. Таким образом, выбрав некоторую случайную вершину, мы сможем добраться от нее до большинства других вершин не более, чем за 6 ребер. Воспользуемся этой эвристикой в нашем алгоритме и выберем вершину наибольшей степени в качестве базовой. И будем обрабатывать два множества различным образом - для меньшего (вершины, которые находятся на расстоянии больше шести) будем запускать параллельный обход в ширину для каждой вершины, для большого - воспользуемся методом динамического программирования для подсчета ответа.

Но прежде, чем приступить к описанию непосредственно алгоритма опишем структуру данных, предложенную Майком Рэйни и Умутом Акаром для обработки множества ребер на нескольких ядрах - Фронтир (Frontier, англ.)

2.4.2. Структура данных Фронтир

Структура данных подробно рассмотрена в статье Умут Акара. Здесь же приведено краткое ее описание, основные принципы работы и интерфейс. Фронтир представляет из себя некоторый набор ребер. При этом он поддерживает операций разделения множества пополам, слияния множеств, добавления ребер вершины и итерирования по ребрам. При этом операций слияния и разбиения выполняются за время пропорциональное O(log n), добавление ребер вершины происходит за константу, а итерирование за константу для каждого ребра. Такая асимптотика достигается за счет лежащей в основе bootstrapped chunked sequence, которая представляет из себя последовательность, где каждому элементу сопоставляется его вес. И операций слияния и разбиения выполняются в соответсвии с этими весами и выполняются за O(log n). Более подробное описание bootstrapped chunked sequence приведено в статье.

2.4.3. Работа алгоритма

Как уже было отмечено ранее, работа алгоритма разбивается на три этапа - анализ графа и выбор базовой вершины, обработка меньшего множества и обработка большего. Разберемся с каждым этапом по отдельности.

Первый и самый простой этап состоит в выборе базовой вершины. В качестве нее будет выбрана вершина с наибольшей степенью. После этого из этой вершины будет запущен обход в ширину, который найдет все вершины, отстоящие не более, чем на K (в описанном в предыдущем пункте случае K = 6). Таким образом, все такие вершины попадают в множество, которое будет обработано на третьем этапе алгоритма. Все остальные вершины попадают в второе множество.

В качестве алгоритма для обработки второго множества запустим параллельный обход в ширину (он же Беллман-Форд) для каждой из вершин этого множества. Кроме того запустим обход для базовой вершины. Эти значения расстоянии нам помогут на третьем этапе алгоритма.

Для поиска искомых значении для вершин третьего множества воспользуемся методом динамического программирования. Но сперва обсудим основные принципы построения алгоритма и доказательство его корректности.

Рассмотрим некоторую вершину, которая находится на расстояний D от базовой вершины (D <= K). То какие вершины для нее могут находится на расстоянии I? Это могут быть только те вершины, которые находятся на расстоянии [I-D, I+D] от базовой. Иначе бы не выполнялось свойство, что путь кратчайший. С другой стороны, если рассмотреть некоторую вершины, расстояние до которой равняется I, то для всех вершин из "большого"множества верно, что кратчайшее расстояние от них до нее варьируется в промежутке [I-K, I+K].

Предположим, что мы запустили обход в ширину из всех вершин большого множества. То какие вершины могут быть в слое с номером I? Ответ вытекает из рассуждений предыдущего абзаца - только вершины, расстояние от которых до базовой варьируется в промежутке [I-K, I+K]. То есть каждая из вершин будет принимать участие в не более, чем 2K+1 слоях. То построим для каждого слоя обхода в ширину множество возможных вершин на этом слое. Это избавит нас от построения следующего фронтира по текущему. И при этом общее количество вершин во всех слоях будет пропорционально числу вершин в графе (если учитывать, что K - небольшое число, меньшее 7).

После того, как мы построили набор вершин для каждого из слоев мы можем воспользоваться структурой Фронтир для эффективного распараллеливания процесса обработки ребер, исходящих из этих вершин. Но к текущему моменту мы никак не воспользовались тем фактом, что вершины расположены близко друг к другу, и, может быть, существует способ для оптимальной обработки группы вершин. Такой подход существует и основан на идее динамического программирования и применения битовых векторов.

Будем поддерживать две динамики. Значениями в полях массива будут битовые вектора - это некоторая структура, где каждый из битов соответсвует вершине из "большого"множества. Первая из динамик mask[u][i] - это битовый вектор вершин, расстояние от которых до U равно I. Вторая из динамик calc[u][i] - набор вершин, расстояние от которых до U не более, чем I.

Рассмотрим процесс пересчета значений динамики. Для подсчета текущего значения mask воспользуемся формулой (2.1). Обратим внимание, что в битовые вектора должны поддерживать битовые логические операции.

$$mask[v][i] = (OR(mask[u][i{-}1] where \exists edge u{-}>v)) AND (NOT calc[v][i{-}1]) \tag{2.1}$$

В свою очередь calc пересчитывается согласно (2.2)

$$calc[v][i] = calc[v][i-1] OR mask[v][i] \tag{2.2}$$

Наконец, как по имеющимся данным динамики восстановить ответ? Для каждого значения mask[u][i] найдем единичные биты в маске. Установленный в единицу бит J говорит о том, что расстояние от вершины из "большого"множества с идентификатором J до вершины u равно i. Таким образом, мы сможем полностью восстановить ответ для каждой вершины.

Итого, псевдокод алгоритма выглядит следующим образом.

Алгоритм 8 Параллельная версия для социальных графов

$K \leftarrow 6$

procedure AllPairsSocialPar(G)
    $dist \leftarrow \{\{\infty...\infty\}...\{\infty...\infty\}\}$
    $handleByBaseVertexSet, otherVertexSet \leftarrow ConstructSets(G)$
    $AllPairsPar1(G, otherVertexSet, dist)$
    $CalculateDistancesForBigSet(G, otherVertexSet, dist)$
    return $dist$

procedure ConstructSets(G)
    $baseVertex \leftarrow$ vertex with max degree
    run serial bfs from $baseVertex$
    $handleByBaseVertexSet \leftarrow$ vertices with dist <= K
    $otherVertexSet \leftarrow G.V \setminus handleByBaseVertexSet$
    return $handleByBaseVertexSet, otherVertexSet$

procedure CalculateDistancesForBigSet($G, vertexSet, dist$)
    if $smth$ **then**
        smth
    else
        smth

2.4.4. Сравнение с наивными версиями

бла-бла-бла.

2.5. Выводы

Всякие разные выводы бла-бла-бла.

Заключение

Текст разный [1].

Источники

[1] Shewchuk J. R. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates // Discrete and Computational Geometry. 1996. Vol. 18. P. 305–363.