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. При этом в большинстве своем 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.

С ростом многопроцессорных архитектур мы получили мощный инструмент для более эффективного расчета искомых расстояний - мы получили возможность запускать эти алгоритмы на нескольких вычислительных ядрах. При этом в контексте с параллельными алгоритмами на графах встал вопрос об эффективном использовании ресурсов системы. Эта задача, однако, не имеет такого высокого разнообразие решений, как в случае однопоточного алгоритма. Именно над этой проблемой я работал - в статье освещены различные версии параллельных алгоритмов поиска кратчайшего пути в графах. 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. При этом во многих алгоритмах будет использоваться современные и высокопроизводительные структуры данных и подходы для параллельной обработки ребер графа.

Все представленные алгоритмы были реализованы и протестированы на различных графовых структурах. При этом описанные подходы продемоснтировали высокую скорость работы на реальных графах, что подтверждает их применимость в реальной жизни.

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 $O(V^2 + E)$, $O(E \log V)$ или $O(V \log V + E)$.

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

```
1: procedure ClassicBellmanFord(G, start)
2:     dist ← {∞...∞}
3:     dist[start] ← 0
4:     for i = 0 to |G.vertices| − 1 do
5:         for e ∈ G.edges do
6:             dist[e.to] ← min(dist[e.to], dist[e.from] + e.w)
7:     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

```
1: procedure BFSBellmanFord(G, start)
2:     dist ← {∞...∞}
3:     dist[start] ← 0
4:     CurrentVertexSet ← {start}                ▷ Набор вершин, расстояние до которых обновилось
5:     NextVertexSet ← ∅
6:     step ← 0
7:     while step < |G.vertices| and not CurrentVertexSet.empty() do
8:         step ← step + 1
9:         NextVertexSet.clear()
10:        for v ∈ CurrentVertexSet do
11:            for e ∈ G.edgesFrom[v] do          ▷ Исходящие ребра из текущей вершины
12:                if dist[e.to] > dist[e.from] + e.w then
13:                    dist[e.to] ← dist[e.from] + e.w
14:                    NextVertexSet.insert(e.to)
15:        CurrentVertexSet ← NextVertexSet
16:    return dist
```

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

Прежде чем приступить к описанию параллельной версий выполним небольшую модификацию алгоритма. Будем для каждой вершины перебирать не исходящие ребра, а входящие. Это ход даст нам одно важное преимущество в контексте параллельных алгоритмов - значение кратчайшего расстояние до каждой вершины теперь может изменять лишь один поток, тогда как раньше могли несколько, что увеличивало потенциальные проблема с гонками за ресурс.

Первая версия алгоритма основана на параллельной обработке всех ребер, исходящих из текущей вершины. Псевдокод, который уже использует идею из предыдущего абзаца, приведен ниже.

Алгоритм 3 Параллельный Беллман-Форд по ребрам вершины

```
1:  procedure BellmanFordPar1(G, start)
2:      dist ← {∞...∞}
3:      dist[start] ← 0
4:      for i = 0 to |G.vertices| − 1 do
5:          changed ← false
6:          for v ∈ G.vertices do
7:              minDist ← dist[v]
8:              parfor e ∈ G.edgesTo[v] do                ▷ Входящие ребра в текущую вершину
9:                  minDist ← min(minDist, dist[e.from] + e.w)
10:             if dist[v] > minDist then
11:                 dist[v] ← minDist                     ▷ Атомарно
12:                 changed ← true
13:         if not changed then
14:             break
15:     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 опять же, как и в предыдущем алгоритме, для каждой вершины будем рассматривать набор ребер, входящих в нее.

Алгоритм 4 Параллельный Беллман-Форд по всем ребрам

```
1: procedure BellmanFordPar2(G, start)
2:     dist ← {∞...∞}
3:     dist[start] ← 0
4:     prefsum ← prefix sum by vertices incoming degree
5:     planMap ← empty map
6:     BuildPlan(prefsum, 0, |G.vertices|, planMap)
7:     for i = 0 to |G.vertices| do
8:         if not ProcessLayer(G, dist, planMap, prefsum, 0, |G.vertices|) then
9:             break
10:    return dist
11:
12: procedure BuildPlan(prefsum, startV, endV, resultMap)        ▷ Функция возвращают структуру (хэш-таблицу), которая по отрезку возвращает его середину
13:    edgesNumber ← prefsum[endV] − prefsum[startV]
14:    if edgesNumber < threshold then
15:        midV ← Бинарным поиском по массиву prefsum находим индекс вершины midV, что prefsum[midV] − prefsum[startV] ≈ prefsum[endV] − prefsum[midV]
16:        resultMap[startV][endV] ← midV
17:        BuildPlan(prefsum, startV, midV, resultMap)
18:        BuildPlan(prefsum, midV, endV, resultMap)
19:    return resultMap
20:
21: procedure ProcessLayer(G, dist, planMap, prefsum, startV, endV)
22:    edgesNumber ← prefsum[endV] − prefsum[startV]
23:    if edgesNumber < threshold then
24:        return ProcessVerticesSequentially(G, dist, startV, endV)
25:    else
26:        midV ← planMap[startV][endV]
27:        changed ← false
28:        changed = changed or ProcessLayer(G, dist, planMap, prefsum, startV, midV)
29:        changed = changed or ProcessLayer(G, dist, planMap, prefsum, midV, endV)
30:        return changed
31:
32: procedure ProcessVerticesSequentially(G, dist, startV, endV)
33:    changed ← false
34:    for v = startV to endV − 1 do
35:        minDist ← dist[v]
36:        for e ∈ G.edgesTo[v] do                    ▷ Входящие ребра в текущую вершину
37:            minDist ← min(minDist, dist[e.from] + e.w)
38:        if dist[v] > minDist then
39:            dist[v] ← minDist                      ▷ Атомарно
40:            changed ← true
41:    return changed
```

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.

Ключевым моментом в алгоритме является использованием структуры данных Frontier. Она подробно рассмотрена в статье Умут Акара. Здесь же приведено краткое ее описание, основные принципы работы и интерфейс. Frontier 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.

Кроме того, в алгоритме используются возможности библиотеки для параллельных вычислений PASL взаимодействия между несколькими потоками. А именно каждый из потоков может понимать нуждаются ли в "помощи"другие потоки. И в случае положительного ответа он может "поделиться"данными для вычислений.

Таким образом, псевдокод алгоритма выглядит следующим образом

Алгоритм 5 Параллельный BFS-подобный Беллман-Форд

```
1: procedure BellmanFordPar3(G, start)
2:     dist ← {∞...∞}
3:     layerForVertex ← {−1... − 1}          ▷ Номер последнего уровня, в котором посетили вершину
4:     dist[start] ← 0
5:     layerForVertex[start] ← 0
6:     Frontier ← {G.edgesFrom(start)}                    ▷ Исходящие ребра текущего множества
7:     for layer = 1 to |G.vertices| do
8:         NextFrontier ← ∅
9:         Frontier ← HandleFrontier(Frontier, NextFrontier, layer, dists, layerForVertex)
10:        if Frontier.empty() then
11:            break
12:    return dist
13:
14: procedure HandleFrontier(CurFrontier, NextFrontier, layer, dists, layerForVertex)
15:    while not CurFrontier.empty() do
16:        if hasIncomingQuery() then
17:            if CurFrontier.nbEdges() ≤ cutoff then
18:                rejectQuery()
19:            else
20:                NewCurFrontier ← ∅
21:                NewNextFrontier ← ∅
22:                CurFrontier.split(NewCurFrontier)
23:                fork2(
                       HandleFrontier(CurFrontier, NextFrontier, layer, dists, layerForVertex),
                       HandleFrontier(NewCurFrontier, NewNextFrontier, layer, dists, layerForVertex));
24:                NextFrontier.split(NewNextFrontier)
25:        Frontier.iterNumber(pollingCutoff, updateFunction(from, to, weight, layer, dists, layerForVertex))
26:
27: procedure UpdateFunction(from, to, weight, layer, dists, layerForVertex, NextFrontier)
28:    if TryToUpdateDistance(to, dists[from] + weight, dists) then
29:        if TryToSetVisited(to, layer, layerForVertex) then
30:            NextFrontier.pushEdgesOf(to)
31:
32: procedure TryToSetVisited(vertex, layer, layerForVertex)
33:    if not layerForVertex[vertex] = layer then
34:        return cas(layerForVertex[vertex], layerForVertex[vertex], layer)
35:    return false
36:
37: procedure TryToUpdateDistance(vertex, candidate, dists)
38:    return writeMin(dists[vertex], candidate)                    ▷ Атомарный минимум
```

1.2.4. Сравнение подходов

Несмотря на то, что все три алгоритма в худшем случае имеют одну асимптотику $O(VE)$, 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). Иными словами, в этом случае алгоритм работает за $O(V + E)$. 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 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, что дает заметное преимущество на разреженных графах по сравнению с Алгоритмом 4, где на каждом шаге обрабатываются все ребра. Однако на плотных графах количество таких вершин значительно, что негативно сказывается на производительностью алгоритма в силу множественных и трудоемких операций с памятью. То есть такой подход показывает себя не с лучшей стороны в подобных графах.

1.2.5. Тестирование

Для подтверждения вышеприведенных замечаний все вышеизложенные подходы были реализованы на основе библиотеки для параллельных вычислений PASL. В качестве языка использовался C++. 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.

Название	Вершины	Ребра	Описание
Complete TS	7071	24995985	Полный граф с fraction = 1 (TopSorted)
Complete +	3162	9995082	Полный граф с положительным весом ребер
Complete -	3162	9995082	Полный граф с случайным весом ребер
BalancedTree F	8388607	8388608	Сбалансирование дерево с fraction = F
SquareGrid +	2499561	4999122	Квадратная решетка с положительными ребрами
SquareGrid -	2499561	4999122	Квадратная решетка с случайными ребрами
RandomSparse 0.5+	2500000	25000000	Случайный разреженный граф с положительными ребрами и fraction = 0.5
RandomSparse 0.5-	2500000	25000000	Случайный разреженный граф с любыми ребрами и fraction = 0.5
RandomSparse 0.96+	2500000	25000000	Случайный разреженный граф с положительными ребрами и fraction = 0.96
RandomDense 0.5+	5000	25000000	Случайный плотный граф с положительными ребрами и fraction = 0.5
RandomDense 0.5-	5000	25000000	Случайный плотный граф с любыми ребрами и fraction = 0.5
RandomDense 0.96+	5000	25000000	Случайный плотный граф с положительными ребрами и fraction = 0.96

fraction - Доля лексикографически отсортированных ребер (ребра вида $V \to V + i$)

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 решения для поиска пути, а после будет приведено решение задачи для социальных неориентированных невзвешенных графов, которое будет сочетать в себе 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 $O(V^3)$. 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

```
1:  procedure FLOYD(G)
2:      dist ← {{∞ ... ∞} ... {∞ ... ∞}}
3:      for e ∈ G.edges do
4:          dist[e.from][e.to] ← e.w
5:
6:      for i = 0 to |G.vertices| do
7:          for u = 0 to |G.vertices| do
8:              for v = 0 to |G.vertices| do
9:                  dist[u][v] ← min(dist[u][v], dist[u][i] + dist[i][v])
10:     return dist
```

2.1.2. Альтернативы

В некоторых случаях оказываются эффективны другие подходы. Например, можно для каждой вершины по отдельности запустить некоторый 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 $O(V^2 \log(V) + VE)$. 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 одновременно. И все нижеперечисленные алгоритмы, как и описанные выше альтернативы, хорошо работают 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. В качестве простого критерия остановки разделения множества для обработки будем использовать некоторую константу $threshold$. Кроме того, важно отметить, что необходимо выбирать наиболее подходящую реализацию параллельного Беллмана-Форда в зависимости от типа графа. К примеру, в случае разреженного графа с положительными ребрами нам подойдет последняя реализация, основанная на параллельном обхода в ширину. Таким образом, псевдокод из себя представляет следующее

Алгоритм 7 Наивная параллельная версия

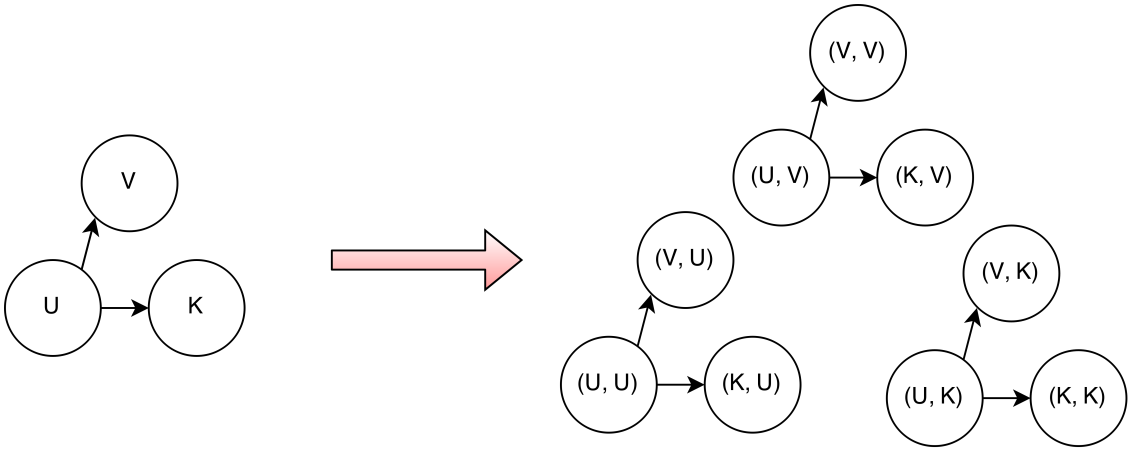
```
1: procedure AllPairsPar1(G)
2:     return HandleVertices(G, 0, |G.vertices|)
3:
4: procedure HandleVertices(G, startVertex, endVertex)
5:     if endVertex − startVertex < threshold then
6:         distances ← run parallel Bellman-Ford for every vertex from startVertex to endVertex (start -
   inclusively, end - exclusively)
7:         return distances
8:     else
9:         midV ← (startVertex + endVertex)/2
10:        fork2(
               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. Описанный алгоритм иллюстрирован на примере простейшего графа из 3 вершин на рисунке 2.1.

Рис. 2.1: Пример преобразования для простейшего графа



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

2.4. Параллельный алгоритм для социальных графов

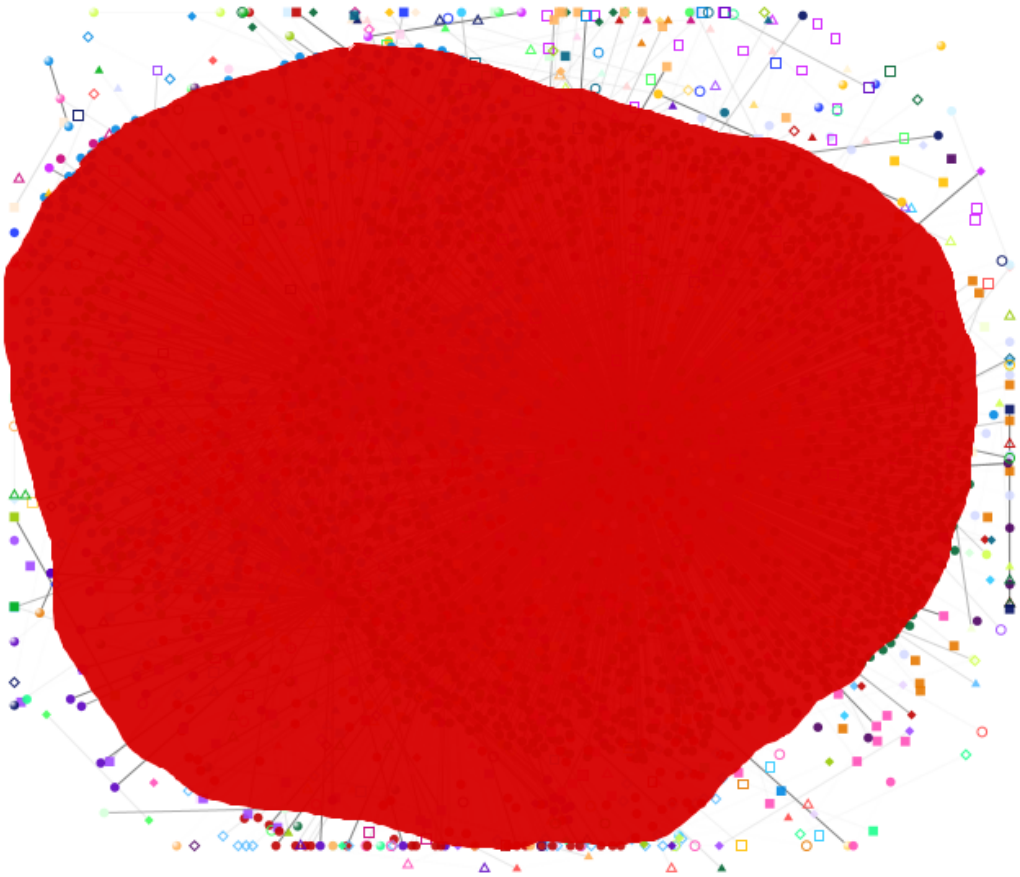
В данной главе будет рассмотрен алгоритм поиска кратчайшего пути между каждой парой вершин для графов реальных социальных сетей. При этом рассмотренный граф будет невзвешенный и неориентированный. Кроме того, в этой главе будет рассмотрена производительность алгоритма на примере реальных подграфов известных социальных сетей, таких как Twitter.

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 рассмотрим два множества - вершины, которые находятся на расстоянии не более 6 от базовой ("большое"множество) и все остальные вершины ("меньшее"множество). Кроме того, будем обрабатывать эти два множества различным образом - для меньшего будем запускать параллельного Беллмана-Форда для каждой вершины, для большого - воспользуемся методом динамического программирования для подсчета ответа. Пример разбиения социального графа на два эти множества проиллюстрирован на рисунке 2.2. Рассмотрим более подробно принцип работы алгоритма.

Рис. 2.2: Пример разбиение социального графа на 2 множества



2.4.2. 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($handleByBaseVertexSet$), которое будет обработано на третьем этапе алгоритма. Все остальные вершины попадают в "меньшее"($otherVertexSet$) множество. Псевдокод этого этапа выглядит следующим образом

Алгоритм 8 Первая фаза алгоритма

```
1:  procedure ConstructSets(G)
2:      baseVertex ← vertex with max degree
3:      dist ← run serial bfs from baseVertex
4:      handleByBaseVertexSet ← ∅
5:      parfor i = 0 to |G.vertices| − 1 do
6:          if dist[i] ≤ K then
7:              handleByBaseVertexSet.add(i)
8:      otherVertexSet ← G.vertices \ handleByBaseVertexSet
9:      otherVertexSet ← otherVertexSet ∪ {baseVertex}
10:     return baseVertex, handleByBaseVertexSet, otherVertexSet
```

В качестве алгоритма для обработки второго множества запустим параллельный обход в ширину (он же Беллман-Форд) для каждой из вершин этого множества. Иными словами, мы просто воспользуемся Алгоритмом 7 для множества вершин (для $otherVertexSet$). При этом исходя из наших эвристик мы предполагаем небольшие размеры множества, что позволяет думать об этом этапе как о значительно менее затратным по времени по сравнению с последним. Кроме того, значения расстояния, посчитан-

ные обходом в ширину из базовой вершины нам помогут на третьем этапе, поэтому добавим в множество $otherVertexSet$ базовую вершину (чтобы в дальнейшем при обращений к глобальному массиву $dist$ в нем содержались корректные расстояния от базовой вершины).

Для поиска искомых значении для вершин множества $handleByBaseVertexSet$ 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 ($d \leq K$). 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 от базовой равняется i, то для всех вершин из множества $handleByBaseVertexSet$ 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 данных Frontier для эффективного распараллеливания процесса обработки ребер, исходящих из этих вершин.

Однако к текущему моменту мы никак не воспользовались тем фактом, что вершины расположены близко друг к другу, и, может быть, существует эффективный технических прием для оптимальной обработки группы вершин. Такой подход существует и основан на идее динамического программирования и применения битовых векторов (bitset). Обратим внимание, что битовые вектора должны поддерживать битовые логические операции(конъюнкция, дизъюнкция и отрицание) и стандартные методы установки соответствующего бита и его получение.

Будем поддерживать две динамики. Значениями в полях массива будут битовые вектора - это некоторая структура, где каждый из битов соответсвует вершине из множества $handleByBaseVertexSet$. 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 $mask[v][i]$ воспользуемся формулой (2.1). Суть формулы состоит в переборе всех ребер графа, входящих в текущую вершину и применение операции битовое или соответствующих масок. При этом мы не должны учитывать данные для тех вершин, расстояние до которых уже посчитано (эта информация хранится в массиве $calc$).

$$mask[v][i] = \neg calc[v][i-1] \wedge \bigvee_{\exists (u,v) \in E} mask[u][i-1] \tag{2.1}$$

В свою очередь $calc$ пересчитывается согласно (2.2)

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

Псевдокод пересчета значений динамики представлен ниже. Обратим внимание, что для каждой вершины нам достаточно хранить всего $2K + 1$ битовых векторов. Однако для упрощения понимания псевдокода

будем обращаться к i слою вершины u просто как $mask[u][i]$, при этом имея в виду эту значительную оптимизацию по памяти. Кроме того другим важным замечанием является то, что когда нам необходимо посчитать значения динамики для слоя i, то нам необходимо обрабатывать предыдущий слой. Таким образом в последнем цикле алгоритма для обработки $layerToCalc$ мы используем $frontierLayer \leftarrow layerToCalc - 1$.

Алгоритм 9 Пересчет динамики

```
1: K ← 6                                          ▷ Максимальная глубина до базовой вершины...
2:
3: procedure CalculateDistancesForBigSet(G, baseVertex, handleByBaseVertexSet)
4:     maxLayer ← calculate max distance from baseVertex
5:     Frontiers ← {Frontier_0 ... Frontier_{maxLayer+K}}        ▷ Фронтир для каждого уровня обхода
6:     VertexSets ← {VertexSet_0 ... VertexSet_{maxLayer+K}}     ▷ Набор вершин для каждого уровня
7:     mask ← {{bitVector(0) ... bitVector(0)} ... {bitVector(0) ... bitVector(0)}}
8:     calc ← {{bitVector(0) ... bitVector(0)} ... {bitVector(0) ... bitVector(0)}}
9:
10:    for i = 0 to maxLayer + K do
11:        for j = dist[baseVertex][i] − K to dist[baseVertex][i] + K do
12:            Frontiers[j].pushEdgesOf(i)
13:            VertexSets[j].addVertex(i)
14:
15:    for v ∈ handleByBaseVertexSet do
16:        mask[v][0] ← bitVector(bitNum(v))      ▷ Будем считать, что существует функция bitNum,
   которая по номеру вершины возвращает соответсвующий ей бит
17:
18:    for layerToCalc = 1 to maxLayer + K do
19:        frontierLayer ← layerToCalc − 1
20:        ProcessLayerLazy(G, Frontiers[frontierLayer], mask, layerToCalc)
21:        parfor v ∈ VertexSets[layerToCalc] do
22:            calc[v][layerToCalc] ← mask[v][layerToCalc]
23:            if layerToCalc is not first layer for vertex v then
24:                calc[v][layerToCalc − 1] ← ¬calc[v][layerToCalc − 1]
25:                mask[v][layerToCalc] ← mask[v][layerToCalc] ∧ calc[v][layerToCalc − 1]
26:                calc[v][layerToCalc − 1] ← ¬calc[v][layerToCalc − 1]
27:                calc[v][layerToCalc] ← calc[v][layerToCalc] ∨ calc[v][layerToCalc − 1]
```

Для полноты алгоритма осталось описать обработку текущего Frontier. Псевдокод этого этапа представлен ниже. В этом алгоритме мы снова (как и в алгоритмах из первой главы) пользуемся интерефейсом, который предоставляет библиотека для параллельных вычислений PASL. Напомню, что для взаимодействие между рабочими потоками существует служебные функции, дающие понять потокам необходимость разбиения

фронтира для обработки на других ядрах или же понять обратное.

Алгоритм 10 Обработка Фронтира

```
1: procedure ProcessLayerLazy(G, Frontier, mask, layer, dists, baseVertex)
2:     while not Frontier.empty() do
3:         if hasIncomingQuery() then
4:             if Frontier.nbEdges() ≤ cutoff then
5:                 rejectQuery()
6:             else
7:                 NewFrontier ← ∅
8:                 Frontier.split(NewFrontier)
9:                 fork2(
                       ProcessLayerLazy(G, Frontier, mask, layer, dists, baseVertex),
                       ProcessLayerLazy(G, NewFrontier, mask, layer, dists, baseVertex));
10:        Frontier.iterNumber(pollingCutoff, updateFunction(mask, from, to, layer, dists, baseVertex))
11:
12: procedure UpdateFunction(mask, from, to, layer, dists, baseV)
13:     if HaveOnLayer(layer - 1, from, dists, baseV) and HaveOnLayer(layer, to, dists, baseV) then
14:         mask[to][layer] ← mask[to][layer] ∨ mask[from][layer − 1]                ▷ Атомарно
15:
16: procedure HaveOnLayer(layer, vertex, dists, baseV)
17:     return |layer − dists[baseV][vertex]| ≤ K
```

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 11 Параллельная версия для социальных графов

```
1: procedure AllPairsSocialPar(G)
2:     dist ← {{∞ ... ∞} ... {∞ ... ∞}}
3:     baseVertex, handleByBaseVertexSet, otherVertexSet ← ConstructSets(G)
4:     AllPairsPar1(G, otherVertexSet, dist)          ▷ Наивный алгоритм для множества вершин
5:     CalculateDistancesForBigSet(G, baseVertex, otherVertexSet, dist)
6:
7:     parfor v = 0 to |G.vertices| do
8:         for j = dist[baseVertex][i] − K to dist[baseVertex][i] + K do
9:             parfor u ∈ handleByBaseVertexSet do
10:                if mask[v][j].getBit(bitNum(u)) = 1 then
11:                    dist[u][v] = j
12:    return dist
```

2.4.3. Сравнение с наивными версиями

Обратим внимание, что асимптотически этот подход не отличается от предыдущих. Однако у него есть ряд преимуществ.

- Пересчет расстояний для группы вершин выполняется быстрее за счет битовых операций. Хоть это и не сказывается на асимптотику, но на практике дает заметный выигрыш.
- Все битовые операции выполняются без выделения дополнительной памяти. При этом изменяются уже созданные поля в массиве для подсчета динамики
- Так как каждый из фронтиров уже построен, то нам не приходится строить следующий фронтир по предыдущему в процессе обработки
- Каждый из получившихся фронтиров довольно большой, что увеличивает его способность к параллелизации
- Все остальные этапы также хорошо параллелятся

Таким образом мы имеем полные основания предполагать, что этот алгоритм на практике покажет заметно лучшие результаты по сравнению с предыдущими версиями.

Для подтверждения этих гипотез все алгоритмы были реализованы и протестированы на той же машине, о которой шла речь в предыдущей главе в контексте Беллмана-Форда.

В качестве графа для тестирования был взят подграф твиттера из 81306 вершинами и 4841532 ребрами. При этом для каждого из ребер считалось неориентированным. Для сравнения алгоритм были взяты два алгоритма - наивная версия и последний описанный алгоритм. Результаты запуска приведен в таблице 2.1.

Алгоритм	Twitter graph
Наивная параллельная версия	427.217
Алгоритм для социальных графов	210.322

Таблица 2.1: Сравнение алгоритмов

Как мы видим мы получили прирост производительности более, чем в два раза, что подтверждает наши ожидания относительно эффективности последнего подхода.

2.5. Выводы

Таким образом в этой части мы описали алгоритм для поиска кратчайшего расстояния на разреженных графах в общем случае (наивная версия) и для частного случая для социальных графов. При этом последний показал свою высокую эффективность на практике на реальных графах социальных сетей.

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.