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

```
1: procedure ClassicBellmanFord(G, start)
2:     dist ← {∞...∞}
3:     dist[start] ← 0
4:     for i = 0 to |G.vertices| do
5:         for e ∈ G.edges do
6:             dist[e.to] ← max(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:      VertexSet ← {start}                    ▷ структура данных для хранения набора вершин
5:      NextVertexSet ← ∅
6:      step ← 0
7:      while step < |G.vertices| and not VertexSet.empty() do
8:          step++
9:          NextVertexSet.clear()
10:         for v ∈ VertexSet 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:         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

```
1:  procedure BellmanFordPar1(G, start)
2:      dist ← {∞...∞}
3:      dist[start] ← 0
4:      for i = 0 to |G.vertices| do
5:          changed ← false
6:          for v ∈ G.vertices do
7:              parfor e ∈ G.fromEdges[v] do
8:                  if dist[e.to] < dist[e.from] + e.w then
9:                      dist[e.to] ← dist[e.from] + e.w
10:                     changed ← true
11:                 dist[e.to] ← max(dist[e.to], dist[e.from] + e.w)
12:         if not changed then
13:             break
14:     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

```
1: procedure BellmanFordPar2(G, start)
2:     dist ← {∞...∞}
3:     dist[start] ← 0
4:     prefsum ← prefix sum of vertices incoming degree
5:     planMap ← BuildPlan(prefsum, 0, |G.vertices|
6:     for i = 0 to |G.vertices| do
7:         if not ProcessLayer(G, planMap, prefsum, 0, |G.vertices|) then
8:             break
9:     return dist
10:
11: procedure BuildPlan(prefsum, startV, endV)        ▷ Функция возвращают структуру, которая по
    отрезку возвращает его середину по количеству ребер
12:     edgesNumber ← prefsum[endV] − prefsum[startV]
13:     if edgesNumber < threshold then
14:         midV ← binary search on edges number
15:         resultMap[startV][endV] ← midV
16:         resultMap[startV][endV].insert(BuildPlan(prefsum, startV, midV)
17:         resultMap[startV][endV].insert(BuildPlan(prefsum, midV, endV)
18:     return resultMap
19:
20: procedure ProcessLayer(G, planMap, prefsum, startV, endV)
21:     edgesNumber ← prefsum[endV] − prefsum[startV]
22:     if edgesNumber < threshold then
23:         process vertices sequentally
24:     else
25:         midV ← planMap[startV][endV]
26:         ProcessLayer(G, planMap, prefsum, startV, midV)
27:         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

```
1:  procedure BellmanFordPar3(G, start)
2:      dist ← {∞...∞}
3:      dist[start] ← 0
4:      Frontier ← {start}   ▷ структура данных для хранения исходящих ребер текущего множества
5:      for i = 0 to |G.vertices| do
6:          Frontier ← handleFrontier(Frontier)      ▷ релаксируем ребра из фронтира и строим новый
7:          if Frontier.empty() then
8:              break
9:      return dist
10:
11: procedure HandleFrontier(Frontier)
12:     recursively divide current frontier, atomically relax edges in frontier and building a new one
13:     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 назна-

чения. 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.

ИСПРАВИТЬ ДАННЫЕ В ТАБЛИЦАХ (раскидать RandomDense и RandomSparse)

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

Таблица 1.1: Bellman-Ford algorithms comparison

Name	Vertices	Edges	Description
Complete TS	1	1	1
Complete sign	1	1	1
BalancedTree fraction	1	1	1
SquareGrid sign	1	1	1
RandomSparse fraction sign	1	1	1
RandomDense fraction sign	1	1	1

Таблица 1.2: Input graph description

1.3. Выводы

TODO

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

В данной главе бла-бла-бла.

2.1. Обзор существующих решений

2.1.1. Алгоритм Флойда

бла-бла-бла.

2.1.2. Альтернативы

бла-бла-бла.

2.2. Наивная параллельная версия

бла-бла-бла.

2.3. Параллельный алгоритм для объединенного графа

бла-бла-бла.

2.4. Параллельный алгоритм для социальных графов

Введение бла-бла-бла.

2.4.1. Идея алгоритма

бла-бла-бла.

2.4.2. Работа алгоритма

бла-бла-бла.

2.4.3. 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.