
Поиск кратчайших путей в графе и его применение

Обзор существующих решений

Сабрина Мусатян
371 группа, Мат-Мех СПБГУ



Обозначения

SSSP(Single-source Shortest Path) - нахождение путей от заданной стартовой вершины до всех остальных.

NSSP(Non-negative Single-source Shortest-Path) - SSSP с условием, что веса всех ребер графа ≥ 0

APSP(All-pair shortest paths) - нахождение кратчайших путей между всеми возможными парами вершин.

Актуальность

- Анализ дорожных сетей
- Анализ медицинских данных
- Анализ компьютерных сетей
- Анализ картографических данных
- Анализ любой информации, представляемой в виде графов

Решения с использованием GPU

- Существенно ускоряют время вычислений
- Облегчают процесс разработки(например, использование CUDA)

Алгоритм Дейкстры



Алгоритм

Вершины могут быть Settled и Unsettled

1. $D[i] = \infty$; $D[s] = 0$. f - frontier node. $f < - s$. Считаем s - settled
2. Для всех v : существует ребро (f, v) и v еще не Settled, делаем релаксацию: если $D[f] + w(f,v) < D[v]$, то обновляем $D[v]$
3. Берем новую вершину и с самым маленьким значением D и которая еще не Settled и считаем ее новым f . Вершина, которая была f до этого, теперь считается Settled.
4. Если все вершины Settled, то алгоритм завершен, иначе шаг 2.

Способы параллелизации:

- Распараллизование “внутренних” операции алгоритма
- Вычисление через непересекающиеся подграфы, результат для которых, может быть вычислен параллельно

Параллельные вычисления внутри алгоритма

- Outer loop: Выбирает вершину для соседей которой будет производится релаксация

Для обеспечения параллельности находится подмножество вершин, для которых можно произвести вычисления параллельно. Основная проблема: вычислить это множество.

- Inner loop: Релаксирует ребра выбранной вершины

Для обеспечения параллельности ребра могут быть релаксированы параллельно.

A New GPU-based Approach to the Shortest Path Problem

Hector Ortega-Arranz, Yuri Torres, Diego R. Llanos Ferraris, Arturo
Gonzalez-Escribano

Основная идея

- Реализация алгоритма Crauser et al. на GPU
- Сравнение данной реализации с CPU и GPU реализациями Martin et al.

Алгоритм Crauser et al.

- A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders, “A parallelization of dijkstra’s shortest path algorithm,” in Mathematical Foundations of Computer Science 1998, ser. LNCS, L. Brim, J. Gruska, and J. Zlatuska, Eds. Springer Berlin / Heidelberg, 1998, vol. 1450, pp. 722–731, 10.1007/BFb0055823. [Online]. Available: <http://dx.doi.org/10.1007/BFb0055823>
- Основано на идее параллельного Outer Loop:

На каждой итерации выбирается множество “безопасных вершин” для которых вычисления могут быть выполнены параллельно.

- Быстрее непараллельного алгоритма ~ в 13 раз

Алгоритм выбора “безопасного множества”

- Precomputation Phase: для всех v из V исходного графа $G(V, E)$ вычисляем:

$$\Delta_{\text{node } v} = \min\{w(v, z) : (v, z) \in E\}$$

- Для всех “Unsettled” (U) вершин в алгоритме Дейкстры на каждой итерации i вычисляем для всех u из U^i :
 $\Delta^i = \min\{\delta(u) + \Delta_{\text{node } u} : u \in U^i\}$, где $\delta(u)$ - расстояние от Source до вершины u на текущей итерации алгоритма
- Тогда “безопасным множеством” F_{i+1} будут вершины для которых: $\delta(v) \leq \Delta^i$

Алгоритм Дейкстры для GPU

- Граф может быть представлен в виде матрицы смежности или списка смежности
- Vector U, содержит в $U[v]$ информацию о том, является ли вершина v - unsettled или нет.
- Vector F, содержит в $F[v]$ информацию о том, входит ли вершина v в текущий Frontier set.
- Vector δ , содержит в $\delta[v]$ текущее расстояние от source до v .

Алгоритм Дейкстры для GPU

```
// CUDA kernels are delimited by <<< ... >>> .  
<<<initialize>>> (U, F, δ); //Initialization  
while (Δ != ∞) do  
    <<<relax>>> (U, F, δ); //Edge relaxation  
    Δ = <<<minimum>>> (U, δ); //Settlement step 1  
    <<<update>>> (U, F, δ, Δ); //Settlement step 2  
end while
```

Relax Kernel

```
tid = thread.Id;  
if (F[tid] == TRUE) then  
    for all j successor of tid do  
        if (U[j] == TRUE) then  
            BEGIN ATOMIC REGION  
                δ[j] = min{δ[j], δ[tid] + w(tid, j)};  
            END ATOMIC REGION  
        end if  
    end for  
end if
```

Для каждой вершины графа заводим отдельную CUDA Thread.
Для всех вершин f , лежащих в F (“безопасное множество”), в CUDA thread происходит процедура релаксации для этой вершины в алгоритме Дейкстры.

Minimum Kernel

- 1) Обновляет значения для U_i : $\delta(u) + \Delta_{node\ u}$
- 2) Находит минимальное такое значение, которое и становится новым значением Δ_i

Update Kernel

```
tid = thread.Id;  
F[tid]= FALSE;  
if (U[tid]==TRUE and δ[tid] <= Δ) then  
    U[tid]= FALSE;  
    F[tid]= TRUE;  
end if
```

Освобождает текущие вершины из F, тем самым они становятся Settled.

Находит вершины, которые будут новым F на шаге i+1

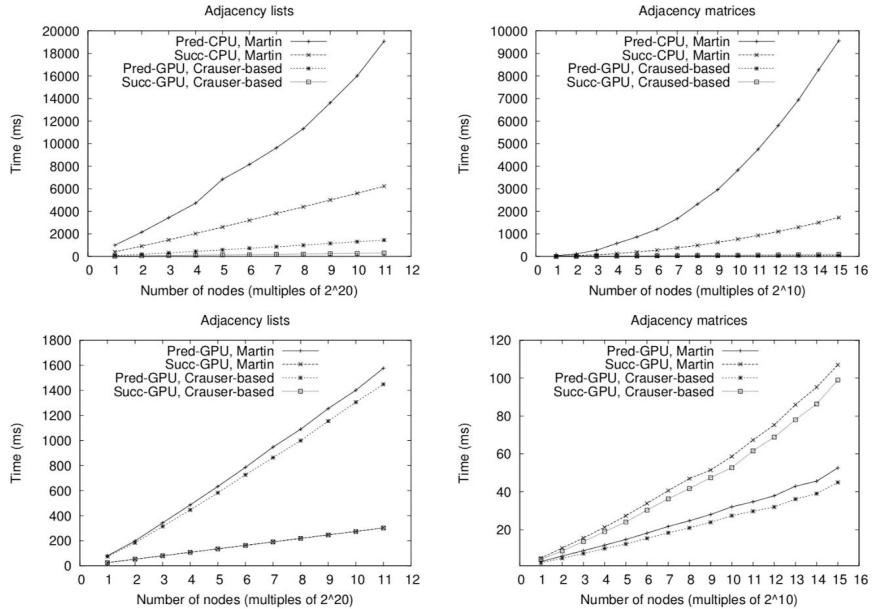
Economic variant

- Использует меньше памяти, но менее мощный
- Вместо массива $\Delta_{node\ v}$ используется единственное значение Δ_{base} равное минимальному ребру в графе.
- Следовательно используется меньше памяти, а также в minimum kernel нет необходимости для каждой вершины считать значение, нужен только минимум по δ .
- Такое решение ведет к тому, что меньшее количество вершин может быть взято во Frontier set, следовательно будет больше непараллельных вычислений.

Алгоритм Martin et al.

- P. Martín, R. Torres, and A. Gavilanes, “CUDA solutions for the SSSP problem,” in Computational Science – ICCS 2009, ser. LNCS, G. Allen, J. Nabrzyski, E. Seidel, G. van Albada, J. Dongarra, and P. Sloot, Eds. Springer Berlin / Heidelberg, 2009, vol. 5544, pp. 904–913,
10.1007/978-3-642-01970-8_91
- Реализует алгоритм дейкстры на GPU
- Successor variant
Выбирает во Frontier set все вершины для которых δ имеет минимальное значение.
- Predecessor variant
В процессе релаксации для каждой unsettled вершины, просматривается есть ли среди ее predecessors вершины, которые находятся во Frontier set. Если находятся такие вершины, то происходит релаксация.

Сравнение алгоритмов



On Ranking Nodes using kNN Graphs, Shortest-paths and GPUs

Ahmed Shamsul Arefin, Regina Berretta, Pablo Moscato

Основные определения

Центральность - определяет относительную значимости элементов в заданной сети. Для графа $G(V,E)$ центральность это такая функция $C: V \rightarrow R$, которая сопоставляет каждой вершине графа число и для любых вершин u и v говорят, что u более центральна, чем v , если $C(u) > C(v)$.

Closeness centrality - для каждой вершины вычисляется по следующей формуле:

$$cc(s) = 1 / \sum_{t \in V} (dist(s, t)), \text{ где } dist(s, t) - \text{длина кратчайшего пути от } s \text{ до } t$$

Eccentricity Centrality - для каждой вершины вычисляется по следующей формуле:

$$ec(s) = 1 / \max_{t \in V} (dist(s, t))$$

Betweenness Centrality - для каждой вершины вычисляется по следующей формуле:

$$bc(s) = \sum_{u \neq s \neq v} (\delta_{uv}(s)), \text{ где}$$

$\delta_{uv}(s) = (\text{кол-во кратчайших путей проходящих через } s \text{ между } u \text{ и } v) / (\text{кол-во кратчайших путей между } u \text{ и } v)$

Существующие исследования в области

- Brandes, U. (2001), 'A faster algorithm for betweenness centrality'
- Edmonds, N., Hoefler, T. & Lumsdaine, A. (2010), A Space-Efficient Parallel Algorithm for Computing Betweenness Centrality in Distributed Memory



Данные

- DNA рака груди
- Данные собраны у 295 пациентов
- Выявлено соответствие между высоко централизованными генами и временем возврата болезни.

kNN Graphs

kNN(k-Nearest Neighbor) Graph - такой граф, где каждая вершина, соединена со своими ближайшими k соседями.

Используется алгоритм GPU-FS-kNN (Arefin, A. S., Riveros, C., Berretta, R. & Moscato, P. (2012b), 'GPU-FS-kNN: A software tool for fast and scalable (knn) computation using GPUs', PLoS ONE, <https://sourceforge.net/p/gpufsknn/>)

Представление графа в памяти GPU

- Из исходных данных формируется kNN Graph.
- Граф хранится в виде массива, где каждый элемент имеет следующие поля: {source, target, weight}.
- Массив отсортирован следующим образом: сначала по полю source, потом по weight.
- Гарантируется, что у каждого source есть в точности k соседей, а значит для каждого source существует ровно k последовательных записей в массиве.

SSSP для kNN graphs

Algorithm 1: Single Source Shortest Path Algorithm

```
Input :  $G_k, s, k$ ;  
Output:  $dist$  array is initialized;  
1 Initialize  $dist[v] \leftarrow -1, \forall v \in V$ ;  
2  $dist[s] \leftarrow 0$ ;  
3  $flag \leftarrow \text{true}$ ;  $d \leftarrow 0$ ;  
4 while  $flag = \text{true}$  do  
5   host  $\rightarrow$  device ( $flag, d$ );  
6   BFS ( $G_k, dist, d, flag, n, k$ ) ;  
7   device  $\rightarrow$  host ( $flag, d$ );  
8    $d \leftarrow d + 1$ ;  
9    $flag \leftarrow \text{false}$   
10 device  $\rightarrow$  host ( $dist$ );  
11 return  $dist$ ;
```

Algorithm 2: BFS (Parallel)

```
Input :  $G_k, dist, d, flag, n, k$ ;  
Output:  $dist$  array is initialized;  
1  $tid \leftarrow$  thread id;  
2 if  $tid < n \times k$  then  
3    $source \leftarrow G_k[tid].source$ ;  
    $target \leftarrow G_k[tid].target$ ;  
   /* Path discovery */  
   if  $dist[source] = d$  then  
      $found \leftarrow \text{true}$ ;  
   else if  $dist[target] = d$  then  
     swap( $target, source$ ) // device  
     function;  
      $found \leftarrow \text{true}$ ;  
   else  $found \leftarrow \text{false}$ ;  
   /* Path Traversal */  
   if  $found = \text{true}$  then  
     if  $dist[target] = -1$  then  
        $dist[target] \leftarrow d + 1$ ;  
        $flag \leftarrow \text{true}$ ;
```

Closeness centrality

Algorithm 3: *kNN Closeness Centrality*

Input : Gk, k ;

Output: Closeness centrality of each vertex stored in cc array;

```
1 Initialize  $dist[t] \leftarrow -1, \forall t \in V$ ;  
2  $flag \leftarrow \text{true}; d \leftarrow 0$ ;  
3 foreach  $s \in V$  do  
4    $dist[s] \leftarrow 0$ ;  
5   while  $flag = \text{true}$  do  
6     host  $\rightarrow$  device ( $flag, d$ );  
7     BFS ( $Gk, dist, d, flag, n, k$ ) ;  
8     device  $\rightarrow$  host ( $flag, d$ );  
9      $d \leftarrow d + 1$ ;  
10     $flag \leftarrow \text{false}$   
11    device  $\rightarrow$  host ( $dist$ );  
12    Accumulate_Closeness ( $dist, n, s, cc$ );  
13 return  $cc$ ;
```

Algorithm 4: *Accumulate_Closeness*

Input : $dist, n, s, cc$;

Output: Closeness centrality of s is accumulated in $cc[s]$;

```
1  $sum \leftarrow 0$ ;  
2  $c = n$ ;  
3 for  $i \leftarrow 0$  to  $n$  do  
4   if  $dist[i] = -1$  then  $c \leftarrow c - 1$ ;  
5   ;  
6   else  $sum \leftarrow sum + dist[i]$ ;  
7   ;  
8   if  $c = 0$  then  $cc[s] \leftarrow 0$ ;  
9   ;  
10  else  $cc[s] \leftarrow ((c - 1)^2 / (n - 1)) / sum$ ;  
11  ;
```

Eccentricity Centrality

Algorithm 5: Accumulate_Eccentricity

Input : $dist, n, s, ec$;

Output: Eccentricity centrality of vertex s is accumulated in $ec[s]$;

```
1 max  $\leftarrow -1$ ;  $e = n$ ;  
2 for  $i \leftarrow 0$  to  $n$  do  
3   if  $dist[i] = -1$  then  $e \leftarrow e - 1$ ;  
4   ;  
5   else if  $dist[i] > max$  then  $max \leftarrow dist[i]$ ;  
6   ;  
7 if  $e = 0$  then  $ec[s] \leftarrow 0$ ;  
8 ;  
9 else  $ec[s] \leftarrow ((e - 1)^2 / (n - 1)) / max$ ;  
10 ;
```



Betweenness Centrality

Brandes, U. (2001), 'A faster algorithm for betweenness centrality', Journal of Mathematical Sociology 25(2), 163–177

$$\delta_{u\bullet}(s) = \sum_{v \in V} \delta_{uv}(s) \quad (5)$$

$$\delta_{u\bullet}(s) = \sum_{w: dist(u,w)=dist(u,s)+1} \frac{\sigma_{us}}{\sigma_{uw}} (1 + \delta_{u\bullet}(w)) \quad (6)$$

$$bc(s) = \sum_{u \neq s \in V} \delta_{u\bullet}(s) \quad (7)$$

Betweenness Centrality

Algorithm 10: Parallel Brandes's (Brandes 2001) Betweenness Centrality 2

```

Input :  $G_k$  and  $k$ ;
Output: Betweenness centrality of each vertex
          stored in  $bc$  array;
1 Initialize  $bc[v] \leftarrow 0$ ,  $\forall v \in V$ ;
2 foreach  $s \in V$  do
3   Single Source Shortest Path ( $G_k, s, k$ )
      (Algorithm 1, see BFS);
4   Initialize  $\delta[v] \leftarrow 0$ ,  $\forall v \in V$ ;
5   /* Backtrace and Accumulate */;
6   while  $d > 1$  do
7     host  $\rightarrow$  device ( $d$ );
8     BFS_Reverse_Order ( $G_k, dist, \sigma, \delta, P, d$ )
        (Algorithm 8);
9     Accumulate_Betweenness
10    ( $s, d, dist, \delta, bc$ ) (Algorithm 9);
11    device  $\rightarrow$  host ( $d$ );
12     $d \leftarrow d - 1$ ;
13
12 device  $\rightarrow$  host ( $bc$ );
13 return  $bc$ ;

```

Algorithm 8: BFS in Reverse Order

```

Input :  $G_k, Pred, \sigma, \delta, n, k$ ;
Output:  $\delta$  is initialized;
1  $tid \leftarrow$  thread id;
2 if  $tid < n \times k$  then
3    $source \leftarrow G_k[tid].source$ ;
4    $target \leftarrow G_k[tid].target$ ;
5   if  $dist[u] = d - 1$  then
6      $found \leftarrow$  true;
7   else if  $dist[w] = d - 1$  then
8     swap( $target, source$ ) // device
       function;
9      $found \leftarrow$  true;
10   else  $found \leftarrow$  false;
11   if  $found =$  true then
12      $bit \leftarrow target \times n + source$ ;
13     if  $P[BIT\_POS(bit)] \& BIT(bit)$  then
14       atomicAdd( $\delta[i], \sigma[i]/\sigma[j] \times (1 + \delta[j])$ );

```

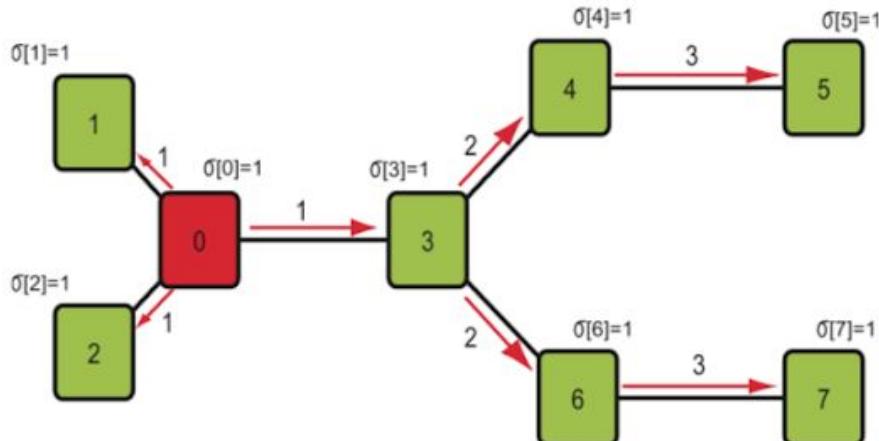
Algorithm 9: Accumulate Betweenness

```

Input :  $bc, \delta$ ;
Output: Betweenness centrality of each vertex
          is stored in  $bc$  array;
1  $tid \leftarrow$  thread id;
2 if  $tid < n$  then
3   if  $tid \neq s$  and  $dist[tid] = d - 1$  then
4      $bc[tid] \leftarrow bc[tid] + \delta[tid]$ ;

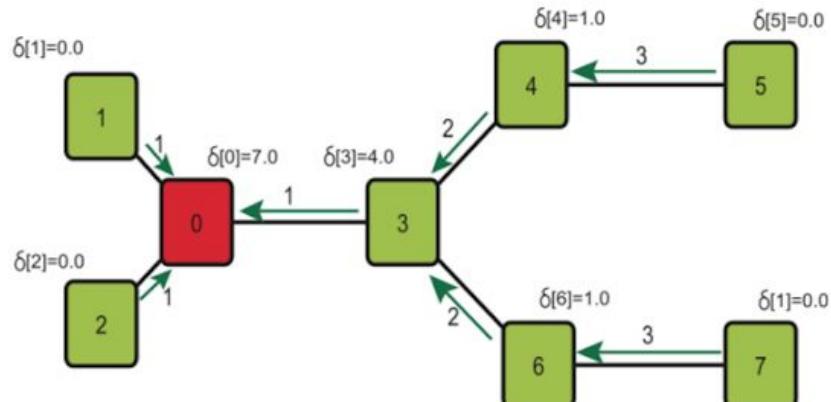
```

Betweenness Centrality



(a) Forward BFS

Betweenness Centrality





Результаты

