Construction of Small Diameter/ASPL Graph with GPU

Hajime Terao

The Graduate School of Informatics and Engineering
The University of Electro-Communications
Graph Golf as optimization problem

- Given:
  - Order of graph: \( n \)
  - Maximum degree of graph: \( d \)

- Minimize:
  - Diameter of graph
  - Average Shortest Path Length (ASPL)

- Note:
  - Diameter has higher priority than ASPL.
  - Smaller Diameter \( \neq \) Better ASPL.
Difficulties in Graph Golf

• Vast search space
  At least $n!$ optimal solutions exist.
• Objective function is not convex.
• One edge can change many shortest paths.
  Every modification to the graph requires entire
  recalculation of ASPL/Diameter.
• The calculation time required for ASPL/Diameter
  is polynomial, but $n$ is so large (up to $1e6$)
My Results

• I found 5 best solutions

<table>
<thead>
<tr>
<th>Order n</th>
<th>Degree d</th>
<th>Diameter</th>
<th>ASPL</th>
<th>ASPL gap</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>4</td>
<td>4</td>
<td>2.64082</td>
<td>0.04898</td>
</tr>
<tr>
<td>1726</td>
<td>30</td>
<td>3</td>
<td>2.47921</td>
<td>0.01834</td>
</tr>
<tr>
<td>9344</td>
<td>6</td>
<td>7</td>
<td>5.48822</td>
<td>0.11436</td>
</tr>
<tr>
<td>65536</td>
<td>6</td>
<td>9</td>
<td>6.73615</td>
<td>0.18302</td>
</tr>
<tr>
<td>100,000</td>
<td>8</td>
<td>7</td>
<td>5.94733</td>
<td>0.20869</td>
</tr>
</tbody>
</table>
My Results

- I found 5 best solutions

<table>
<thead>
<tr>
<th>Order n</th>
<th>Degree d</th>
<th>Diameter</th>
<th>ASPL</th>
<th>ASPL gap</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>4</td>
<td>4</td>
<td>2.6</td>
<td></td>
</tr>
<tr>
<td>1726</td>
<td>30</td>
<td>3</td>
<td>2.47921</td>
<td>0.01834</td>
</tr>
<tr>
<td>9344</td>
<td>6</td>
<td>7</td>
<td>5.48822</td>
<td>0.11436</td>
</tr>
<tr>
<td>65536</td>
<td>6</td>
<td>9</td>
<td>6.73615</td>
<td>0.18302</td>
</tr>
<tr>
<td>100,000</td>
<td>8</td>
<td>7</td>
<td>5.94733</td>
<td>0.20869</td>
</tr>
</tbody>
</table>

Deepest Improvement
Approach

• Find better solution by Simulated Annealing from multiple initial solutions.
• Design symmetric and memory-efficient graph, in order to,
  1. Reduce theoretical calculation time
  2. Avoid memory bandwidth bottleneck
• Make use of GPU for ASPL calculation, achieved about 700x faster than single thread naive CPU implementation
Design of Graph

- I designed “part shift graph”, similar to Cayley Graph
- Vertices have indices $0 \ldots n - 1$
- Indices are regarded as elements of cyclic group $\mathbb{Z}_n$
- Show an example of $(n, d) = (12, 4)$
- Choose size of “part” $m = 3$ from divisors of $n$
- Then construct a “part”.

Indices are regarded as elements of cyclic group $\mathbb{Z}_n$
"part" is a subgraph, all edges join \(0 \ldots m - 1\) vertices.
Design of Graph

- Copy & Shift the part by $m( = 3)$
Design of Graph

• Copy & Shift the part by $m( = 3)$
Design of Graph

• Copy & Shift the part by \( m( = 3) \)
Design of Graph

- Erase duplicated edges

Diameter: 2
ASPL: 1.63636
This graph is symmetric, thereby All Pairs Shortest Path problem can be solved by Single Source Shortest Path (SSSP) problem $m$ times.
Design of Graph

- This graph is symmetric, thereby edge data requires only $O(md)$ space.
Design of Graph

- $m$ is such a small number that $O(md)$ edge data can be stored in the cache.

<table>
<thead>
<tr>
<th>Order $n$</th>
<th>Degree $d$</th>
<th>Diameter</th>
<th>ASPL</th>
<th>ASPL gap</th>
<th>$m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>4</td>
<td>4</td>
<td>2.64082</td>
<td>0.04898</td>
<td>50</td>
</tr>
<tr>
<td>1726</td>
<td>30</td>
<td>3</td>
<td>2.47921</td>
<td>0.01834</td>
<td>2</td>
</tr>
<tr>
<td>9344</td>
<td>6</td>
<td>7</td>
<td>5.48822</td>
<td>0.11436</td>
<td>16</td>
</tr>
<tr>
<td>65536</td>
<td>6</td>
<td>9</td>
<td>6.73615</td>
<td>0.18302</td>
<td>64</td>
</tr>
<tr>
<td>100,000</td>
<td>8</td>
<td>7</td>
<td>5.94733</td>
<td>0.20869</td>
<td>(4,5)</td>
</tr>
<tr>
<td>4855</td>
<td>15</td>
<td>4</td>
<td>3.42917</td>
<td>0.13066</td>
<td>5</td>
</tr>
<tr>
<td>1,000,000</td>
<td>32</td>
<td>5</td>
<td>4.33066</td>
<td>0.34858</td>
<td>64</td>
</tr>
</tbody>
</table>
Simulated Annealing: Overview

• I’m a beginner of Simulated Annealing
• Representation of solution: “part”
• Cooling Schedule: Exponential Cooling
• Initial Temperature: Determined by experiments
• # of iteration: Determined by experiments (3M~10M)
• Energy: Difference of ASPLs
Simulated Annealing: Initial Solution

- “part” can be represented as weighted order $m$ graph

- Generate random graph and convert it into a “part”
- Weight is difference of indices.
Simulated Annealing: The Neighbors of State

• 2 types of the neighbor
  1. Modify a weight of single edge

\[(\text{edge weight}) \% m = (\text{difference of indices}) \% m\]
Simulated Annealing: The Neighbors of State

- 2 types of the neighbor
  2. Cut 2 edges and reconstruct 2 edges

\[(\text{edge weight}) \% m = (\text{difference of indices}) \% m\]
ASPL Calculation

- Average Shortest Path Length (ASPL) of Graph \((V, E)\) is defined as

\[
\text{ASPL}(V, E) = \frac{\sum_{u \in V} \sum_{v \in V} d(u, v)}{n(n - 1)}
\]

- The calculation of All Pairs Shortest Path is needed.
- Calculation time is enormous
  - Floyd–Warshall algorithm: \(O(n^3)\)
  - Solving SSSP for all vertices: \(O(n^2d)\)
  - Solving SSSP for all vertices of the “part”: \(O(nmd)\)
- Parallelization is required
Why GPU Acceleration?

- **Parallelization** is required
- Somehow, my laboratory PC has **two** Geforce GTX 780
- I got **four** PCs equipping Geforce GTX745

OS: Ubuntu18.04  
CPU: i7-4790 3.6GHz  
RAM: 8GB  
GPU: Geforce GTX745  

- I implemented CUDA code inspired by Beamer’s Bottom-up Algorithm
Find **unvisited** vertices from each vertex in the **frontier**

- **Frontier**: 0
- **Next**: 3 2 4

![Diagram](image-url)
Parallel Top-Down BFS

Find **unvisited** vertices from each vertex in the **frontier**

- **Frontier:** 3 2 4
- **Next:** 5 6 7

Diagram: Shows a graph with vertices labeled 0 to 7, and edges connecting them. The vertices are color-coded as follows:
  - Purple: Frontier
  - Orange: Visited
  - Green: Unvisited
Parallel Top-Down BFS

Find **unvisited** vertices from each vertex in the **frontier**
Parallel Top-Down BFS

Find **unvisited** vertices from each vertex in the **frontier**

Atomic operations needed

Wasted edge traversals

- **frontier**
- **visited**
- **unvisited**
Find **frontier** vertices from each **unvisited** vertex

Beamer’s Algorithm (Bottom-up BFS)

```
frontier
0
3 2 4
next
```

- **frontier** vertices
- **visited** vertices
- **unvisited** vertices
Beamer’s Algorithm (Bottom-up BFS)

Find **frontier** vertices from each **unvisited** vertex

```
3 2 4
```

```
5 6 7
```

- 0
- 1
- 2
- 3
- 4
- 5
- 6
- 7

**Frontier**: 3 2 4

**Next**: 5 6 7

- **Frontier**: vertices in the current layer that have not been visited.
- **Visited**: vertices that have been visited.
- **Unvisited**: vertices that have not been visited yet.
Beamer’s Algorithm (Bottom-up BFS)

Find **frontier** vertices from each **unvisited** vertex

```plaintext
<table>
<thead>
<tr>
<th>3</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>frontier</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
<table>
<thead>
<tr>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>next</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

No atomic operation needed

Wasted edge traversals reduced

- **frontier**
- **visited**
- **unvisited**
Beamer’s Algorithm (Bottom-up BFS)

Find **frontier** vertices from each **unvisited** vertex
Optimization for CUDA

• Beamer’s Bottom-up Algorithm is less efficient for ASPL calculation with CUDA because
  1. visited/unvisited flag is 1 bit information, but each edge traversal cause 32 byte memory access. Memory bandwidth limits performance.
  2. branch divergence gets most of the CUDA cores assigned to vertices idle.
• To maximize efficiency, perform multiple SSSP at once.
Optimization for CUDA

Assign bit vectors for all vertices
Set \( i \) th bit of \( i \) th vector “1”
Update vectors with bitwise OR of neighbors

iter : 1

○ 0 (unvisited)  ● 1 (visited)  ● 1 (updated)
Optimization for CUDA

Update vectors in parallel

iter : 1

- \( B_0 \) (unvisited)
- \( B_1 \) (visited)
- \( B_2 \) (updated)
- \( B_3 \) (unvisited)
- \( B_4 \) (updated)
- \( B_5 \) (visited)
- \( B_6 \) (unvisited)
- \( B_7 \) (visited)
In $t$ th iteration, # of red corresponds # of distance $t$ pairs

iter : 1
# red : 24

Optimization for CUDA
Update with bitwise OR of neighbors

iter : 2

# : 24
Update with bitwise OR of neighbors

iter : 3
# : 8
Optimization for CUDA

Terminate iteration when all bit get "1"

iter : 4
# : 0

0 (unvisited)  1 (visited)  1 (updated)
Optimization for CUDA

iter : 1
# : 24

iter : 2
# : 24

iter : 3
# : 8

ASPL : \[
\frac{24 \cdot 1 + 24 \cdot 2 + 8 \cdot 3}{8 \cdot 7} = 1.714285\ldots
\]
Performance of the implementation

- This implementation reduces memory access drastically.
- ASPL of part shift graph with \((n, d, m) = (1e6, 32, 64)\) can be calculated in 113ms with Geforce GTX780.
- ASPL of entire graph with \((n, d) = (1e6, 32)\) is calculated in 160s, \(\mathbf{710x}\) faster than native serial BFS implementation with i7-8700.
Conclusion

• “part shift graph” can achieve small Diameter/ASPL.
• GPU acceleration is powerful tool for ASPL calculation of large graphs.
• Source Code

https://github.com/confused-uec/graphgolf-cuda

• References


Scott Beamer, Understanding and Improving Graph Algorithm Performance, Technical Report UCB/EECS-2016-153 EECS Department, University of California, Berkeley 2016.