Speedup Graph Processing by Graph Ordering

Presenter: Haonan Wang

haonanw@mit.edu

April 23, 2019

Overview

1. Background
    - Motivation
    - Graph Access Patterns
2. Algorithm
    - The GO Algorithm
    - GO-PQ
3. Evaluation

Section 1

Background

Subsection 1

Motivation

Motivation

Graphs are important

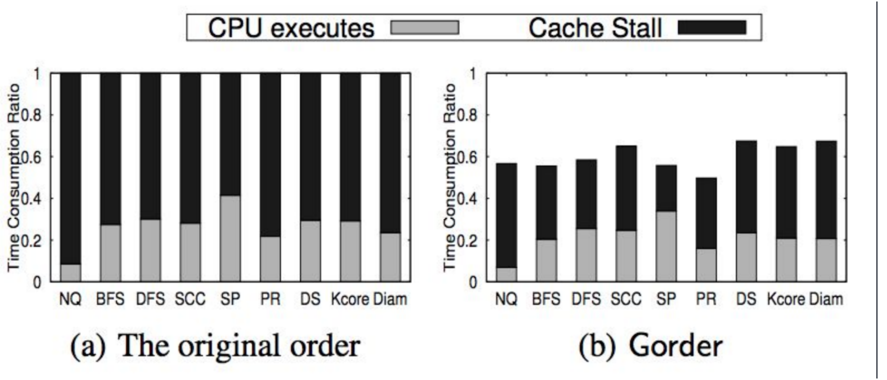
- Graph is a popular data model for Big Data
  social graph, web graph, knowledge graph
- Large graph is emerging with increasing size
  Friendship connection in Facebook, Twitter, ...

CPU cache performance is key issue in efficiency in DBS

- cache miss latency takes a half of the execution time in database systems

Design general optimization approach for graph model

The cost of cache miss



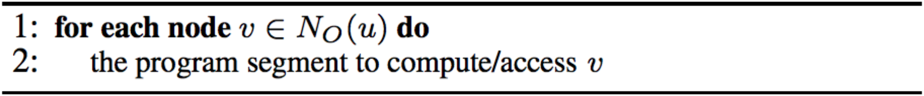
(a) The original order

(b) Gorder

Subsection 2

Graph Access Patterns

Common Ideas About Graph Access



```
for each node $v \in N_O(u)$ do
    the program segment to compute/access $v$
```

Common Relationships: Neighbours / Sibling

Key Idea here is that sibling relationship is a dominating factor.

$$\binom{d_O(u)}{2} \gg d_O(u)$$

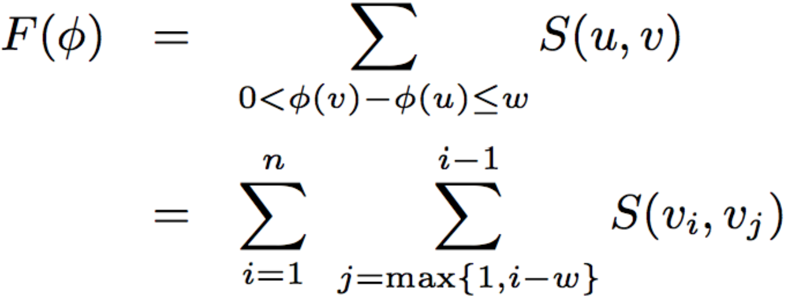
Measure The Problem With Function

- Measure the Closeness of two points:

$$S(u, v) = S_s(u, v) + S_n(u, v)$$

- Our problems turns into a permutation problem!
-

Function.png

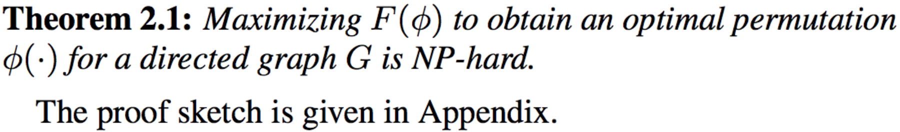


$$F(\phi) = \sum_{0<\phi(v)-\phi(u)\leq w} S(u, v) = \sum_{i=1}^{n} \sum_{j=\max\{1, i-w\}}^{i-1} S(v_i, v_j)$$

Section 2

Algorithm

Some Key Ideas About The Algorithms

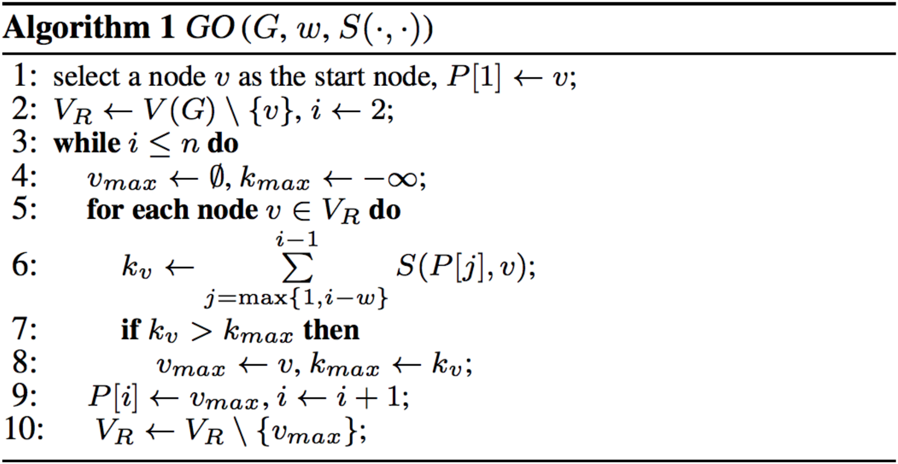


Theorem 2.1: *Maximizing $F(\phi)$ to obtain an optimal permutation $\phi(\cdot)$ for a directed graph G is NP-hard.*

The proof sketch is given in Appendix.

- All algorithms here are all approximate algorithms.
- If we set $w = 1$, the problem here is actually equivalent to the maximum traveling salesman problem, denoted as maxTSP for short.

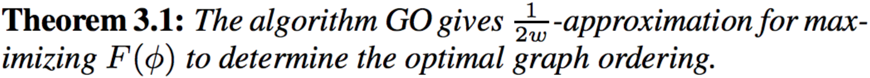
Go Algorithm(Simple Greedy One)



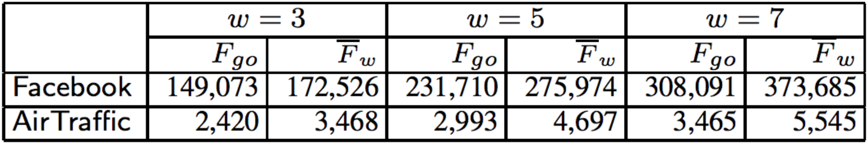
Algorithm 1 $GO(G, w, S(\cdot,\cdot))$

```
select a node $v$ as the start node, $P[1] \leftarrow v$;
$V_R \leftarrow V(G) \setminus \{v\}$, $i \leftarrow 2$;
while $i \le n$ do
    $v_{max} \leftarrow \emptyset$, $k_{max} \leftarrow -\infty$;
    for each node $v \in V_R$ do
        $k_v \leftarrow \sum_{j=\max\{1,i-w\}}^{i-1} S(P[j], v)$;
        if $k_v > k_{max}$ then
            $v_{max} \leftarrow v$, $k_{max} \leftarrow k_v$;
    $P[i] \leftarrow v_{max}$, $i \leftarrow i + 1$;
    $V_R \leftarrow V_R \setminus \{v_{max}\}$;
```

Good Performance Of Go Algorithm



Theorem 3.1: *The algorithm GO gives $\frac{1}{2w}$-approximation for maximizing $F(\phi)$ to determine the optimal graph ordering.*



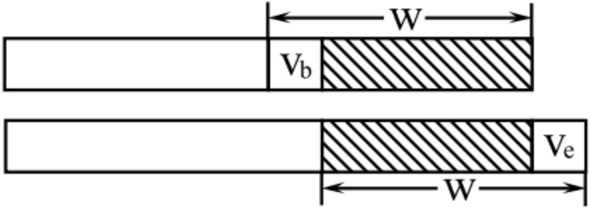
	$w=3$		$w=5$		$w=7$	
	F_{go}	$\overline{F}_w$	F_{go}	$\overline{F}_w$	F_{go}	$\overline{F}_w$
Facebook	149,073	172,526	231,710	275,974	308,091	373,685
AirTraffic	2,420	3,468	2,993	4,697	3,465	5,545

Table 1: F_{go} **and** $\overline{F}_w$

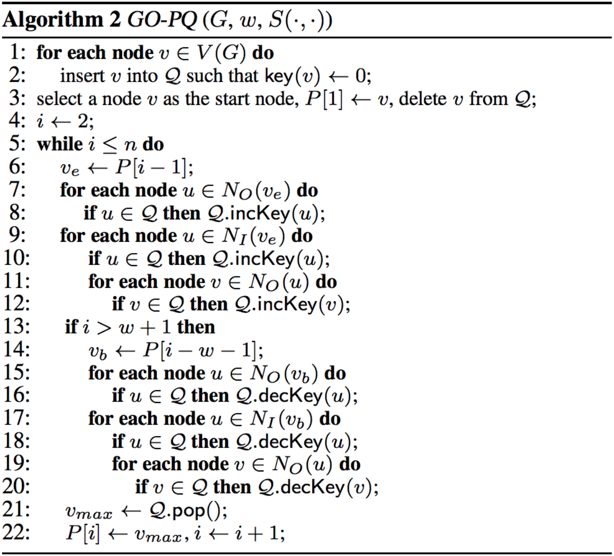
The Priority Queue based Algorithm

- Go is expensive.
  Time complexity:
  $$O(w \cdot d_{max} \cdot n^2)$$
- Reasons for the inefficiency
  (1)Repeatedly computes score function w times for the same pair $(v_j\ ,\ v)$ while v_j is in the window of size w (2)it scans every node v in the set of remaining nodes VR in every iteration

Explain For What we do in the new algorithm



GOPQ algorithm



Algorithm 2 *GO-PQ* $(G, w, S(\cdot,\cdot))$

```
for each node $v \in V(G)$ do
    insert $v$ into $\mathcal{Q}$ such that key$(v) \leftarrow 0$;
select a node $v$ as the start node, $P[1] \leftarrow v$, delete $v$ from $\mathcal{Q}$;
$i \leftarrow 2$;
while $i \leq n$ do
    $v_e \leftarrow P[i-1]$;
    for each node $u \in N_O(v_e)$ do
        if $u \in \mathcal{Q}$ then $\mathcal{Q}$.incKey$(u)$;
    for each node $u \in N_I(v_e)$ do
        if $u \in \mathcal{Q}$ then $\mathcal{Q}$.incKey$(u)$;
        for each node $v \in N_O(u)$ do
            if $v \in \mathcal{Q}$ then $\mathcal{Q}$.incKey$(v)$;
    if $i > w + 1$ then
        $v_b \leftarrow P[i-w-1]$;
        for each node $u \in N_O(v_b)$ do
            if $u \in \mathcal{Q}$ then $\mathcal{Q}$.decKey$(u)$;
        for each node $u \in N_I(v_b)$ do
            if $u \in \mathcal{Q}$ then $\mathcal{Q}$.decKey$(u)$;
            for each node $v \in N_O(u)$ do
                if $v \in \mathcal{Q}$ then $\mathcal{Q}$.decKey$(v)$;
    $v_{max} \leftarrow \mathcal{Q}$.pop();
    $P[i] \leftarrow v_{max}$, $i \leftarrow i + 1$;
```

Implementation Of the Priority Queue

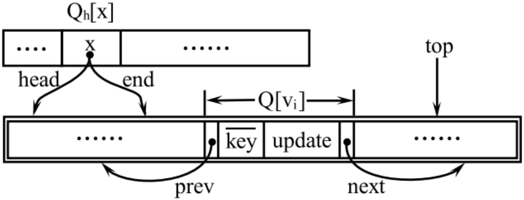
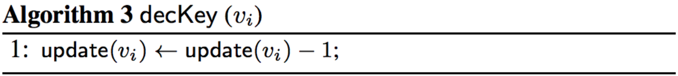


Figure 7: The Priority Queue: $\mathcal{Q}$**,** $\mathcal{Q}_h$**, and** top

Implementation Of the Priority Queue

- Lazy Update, use $\bar{key}$ and update instead of key
- Maintain the double link list until there should be changes
- Use the head and end to get the position for the first element and the last element of the subqueue with same key

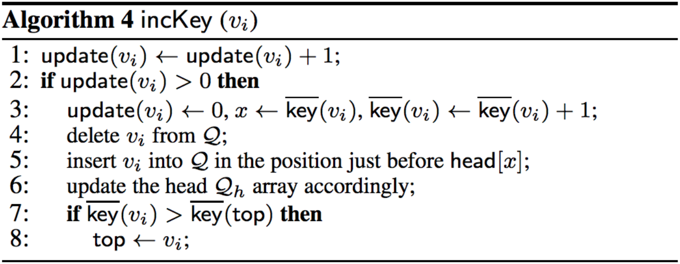
Implementation Of the Priority Queue



Algorithm 3 decKey (v_i)

```
update($v_i$) ← update($v_i$) $-$ 1;
```

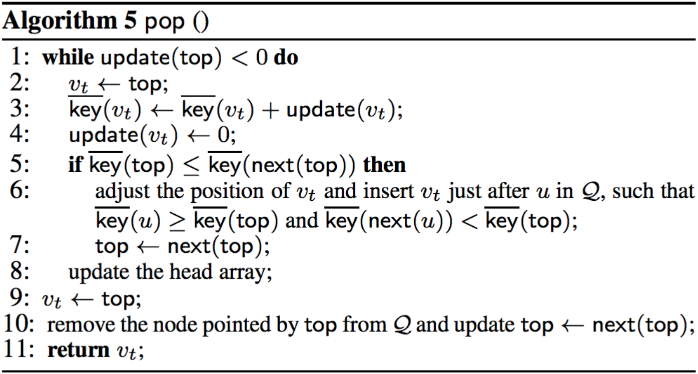
Implementation Of the Priority Queue



Algorithm 4 incKey (v_i)

```
update(v_i) ← update(v_i) + 1;
if update(v_i) > 0 then
    update(v_i) ← 0, $x ← \overline{key}(v_i)$, $\overline{key}(v_i) ← \overline{key}(v_i) + 1$;
    delete $v_i$ from $\mathcal{Q}$;
    insert $v_i$ into $\mathcal{Q}$ in the position just before head[$x$];
    update the head $\mathcal{Q}_h$ array accordingly;
    if $\overline{key}(v_i) > \overline{key}$(top) then
        top ← $v_i$;
```

Implementation Of the Priority Queue



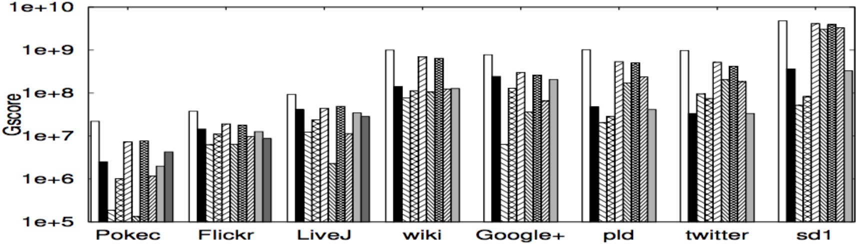
Algorithm 5 pop ()

```
while update(top) $< 0$ do
    $v_t \leftarrow$ top;
    $\overline{\text{key}}(v_t) \leftarrow \overline{\text{key}}(v_t) + \text{update}(v_t)$;
    update$(v_t) \leftarrow 0$;
    if $\overline{\text{key}}(\text{top}) \leq \overline{\text{key}}(\text{next}(\text{top}))$ then
        adjust the position of $v_t$ and insert $v_t$ just after $u$ in $\mathcal{Q}$, such that
        $\overline{\text{key}}(u) \geq \overline{\text{key}}(\text{top})$ and $\overline{\text{key}}(\text{next}(u)) < \overline{\text{key}}(\text{top})$;
        top $\leftarrow$ next(top);
    update the head array;
$v_t \leftarrow$ top;
remove the node pointed by top from $\mathcal{Q}$ and update top $\leftarrow$ next(top);
return $v_t$;
```

Section 3

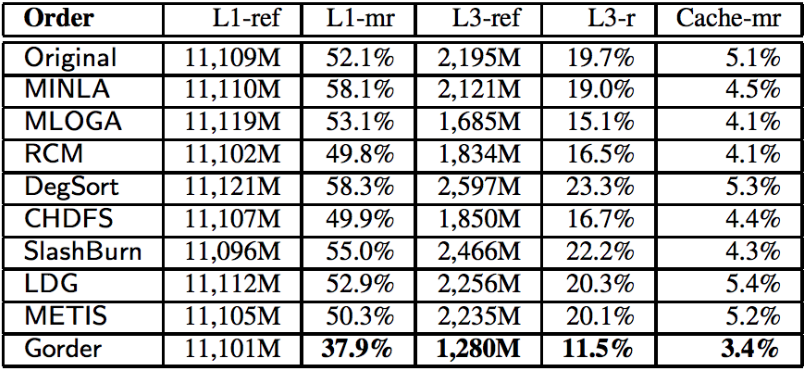
Evaluation

Implementation Of the Priority Queue



(a) $F(\cdot)$ by Different Orderings

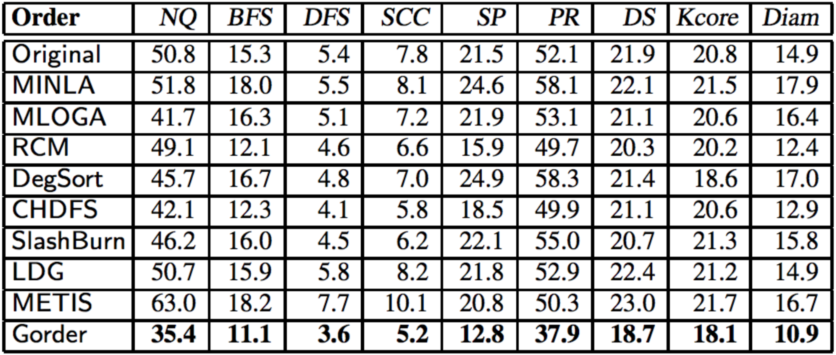
Implementation Of the Priority Queue



Order	L1-ref	L1-mr	L3-ref	L3-r	Cache-mr
Original	11,109M	52.1%	2,195M	19.7%	5.1%
MINLA	11,110M	58.1%	2,121M	19.0%	4.5%
MLOGA	11,119M	53.1%	1,685M	15.1%	4.1%
RCM	11,102M	49.8%	1,834M	16.5%	4.1%
DegSort	11,121M	58.3%	2,597M	23.3%	5.3%
CHDFS	11,107M	49.9%	1,850M	16.7%	4.4%
SlashBurn	11,096M	55.0%	2,466M	22.2%	4.3%
LDG	11,112M	52.9%	2,256M	20.3%	5.4%
METIS	11,105M	50.3%	2,235M	20.1%	5.2%
Gorder	11,101M	**37.9%**	**1,280M**	**11.5%**	**3.4%**

Table 3: Cache Statistics by *PR* over Flickr (M = Millions)

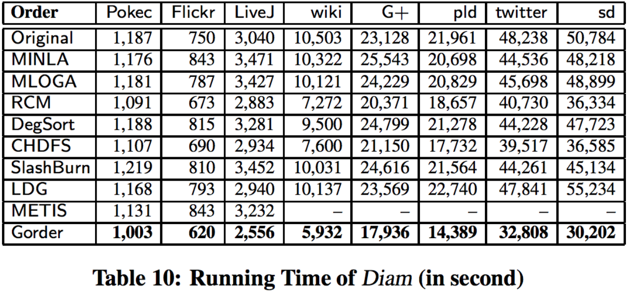
Implementation Of the Priority Queue



Order	*NQ*	*BFS*	*DFS*	*SCC*	*SP*	*PR*	*DS*	*Kcore*	*Diam*
Original	50.8	15.3	5.4	7.8	21.5	52.1	21.9	20.8	14.9
MINLA	51.8	18.0	5.5	8.1	24.6	58.1	22.1	21.5	17.9
MLOGA	41.7	16.3	5.1	7.2	21.9	53.1	21.1	20.6	16.4
RCM	49.1	12.1	4.6	6.6	15.9	49.7	20.3	20.2	12.4
DegSort	45.7	16.7	4.8	7.0	24.9	58.3	21.4	18.6	17.0
CHDFS	42.1	12.3	4.1	5.8	18.5	49.9	21.1	20.6	12.9
SlashBurn	46.2	16.0	4.5	6.2	22.1	55.0	20.7	21.3	15.8
LDG	50.7	15.9	5.8	8.2	21.8	52.9	22.4	21.2	14.9
METIS	63.0	18.2	7.7	10.1	20.8	50.3	23.0	21.7	16.7
Gorder	**35.4**	**11.1**	**3.6**	**5.2**	**12.8**	**37.9**	**18.7**	**18.1**	**10.9**

Table 6: L1 Cache Miss Ratio on Flickr **(in percentage %)**

Implementation Of the Priority Queue



Order	Pokec	Flickr	LiveJ	wiki	G+	pld	twitter	sd
Original	1,187	750	3,040	10,503	23,128	21,961	48,238	50,784
MINLA	1,176	843	3,471	10,322	25,543	20,698	44,536	48,218
MLOGA	1,181	787	3,427	10,121	24,229	20,829	45,698	48,899
RCM	1,091	673	2,883	7,272	20,371	18,657	40,730	36,334
DegSort	1,188	815	3,281	9,500	24,799	21,278	44,228	47,723
CHDFS	1,107	690	2,934	7,600	21,150	17,732	39,517	36,585
SlashBurn	1,219	810	3,452	10,031	24,616	21,564	44,261	45,134
LDG	1,168	793	2,940	10,137	23,569	22,740	47,841	55,234
METIS	1,131	843	3,232	–	–	–	–	–
Gorder	**1,003**	**620**	**2,556**	**5,932**	**17,936**	**14,389**	**32,808**	**30,202**

Table 10: Running Time of *Diam* (in second)

The End