WONDERLAND
A NOVEL ABSTRACTION-BASED OUT-OF-CORE
GRAPH PROCESSING SYSTEM

Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, Kang Chen
Presented by Divya Gopinath
6.886 Spring 2019
AGENDA

- Motivation
- Background: out-of-core processing & graph abstraction
- System implementation
- Case study & benchmarking
- Conclusion
- Discussion
• Motivation
• **Background: out-of-core processing & graph abstraction**
  • System implementation
  • Case study & benchmarking
  • Conclusion
  • Discussion
Graphs are everywhere, and require iterative algorithms to process input graph.

- How can we design an algorithm that isn’t fully in-memory or distributed? **Out-of-core**
- How can we reduce graph size? **Abstraction**
OUT-OF-CORE PROCESSING

• Equivalent to **disk-based single-machine** systems (external memory model)
  • Small portion of graph in main memory
  • Spill remainder to disk
• Shift from reducing slow random disk I/O to improving convergence speed
• Improve *execution efficiency* of algorithm
• Concise lossy representation of original graph
• Bootstrap initial result to accelerate convergence
• Fit abstraction in memory
How do we apply the out-of-core model to a graph processing framework that uses abstraction in some way?
How do we apply the out-of-core model to a graph processing framework that uses abstraction in some way?

**Challenge 1: Performance**
Most existing techniques assume original graph can be contained in memory—how do we adapt this for out-of-core systems? Transformations can be complex of need multiple passes through input data.
BACKGROUND

How do we apply the **out-of-core model** to a **graph processing** framework that uses **abstraction** in some way?

**Challenge 1: Performance**
Most existing techniques assume original graph can be contained in memory—how do we adapt this for out-of-core systems? Transformations can be complex of need multiple passes through input data.

**Challenge 1I: Programmability**
Adding vertices/edges not in the original graph to create the abstraction makes programming harder—may require application-specific algorithms to recover the initial result.
BACKGROUND
OUT-OF-CORE GRAPH PROCESSING

**GraphChi**  Vertex-centric

<table>
<thead>
<tr>
<th>Interval(1)</th>
<th>Interval(2)</th>
<th>Interval(3)</th>
<th>…</th>
</tr>
</thead>
</table>

**X-Stream**  Edge-centric

- **Shuffle()** – external sort to provide updates
- **Scatter()** – read all edges and output updates as stream
- **Gather()** – apply updates
**Background**

**Out-of-Core Graph Processing**

**Grid graph**  Partition based

- Partition edges into \( P \times P \) grids
- Each partition contains vertices within contiguous range
- Define edge function that updates \( \text{dst} \) from \( \text{src} \)

<table>
<thead>
<tr>
<th></th>
<th>P = 2</th>
<th>Vertex 1</th>
<th>Vertex 2</th>
<th>Vertex 3</th>
<th>Vertex 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(1, 2)</td>
<td>(2, 1)</td>
<td>(1, 3)</td>
<td>(2, 4)</td>
<td></td>
</tr>
<tr>
<td>Vertex 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vertex 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vertex 3</td>
<td>(3, 2)</td>
<td></td>
<td></td>
<td>(4, 3)</td>
<td></td>
</tr>
<tr>
<td>Vertex 4</td>
<td>(4, 2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Grid graph**  
Partition based

- Partition edges into $P \times P$ grids
- Each partition contains vertices within contiguous range
- Define edge function that updates $dst$ from $src$

**Common patterns:**
Disjoint partitions
Fixed processing order

---

**Table:**

<table>
<thead>
<tr>
<th>$P = 2$</th>
<th>Vertex 1</th>
<th>Vertex 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex 1</td>
<td>(1, 2)</td>
<td>(1, 3)</td>
</tr>
<tr>
<td>Vertex 2</td>
<td>(2, 1)</td>
<td>(2, 4)</td>
</tr>
<tr>
<td>Vertex 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vertex 4</td>
<td>(3, 2)</td>
<td>(4, 3)</td>
</tr>
<tr>
<td></td>
<td>(4, 2)</td>
<td></td>
</tr>
</tbody>
</table>
Create abstractions that consist only of vertices and edges existing in original graph (select subset of edge set)

Enforces application-independent abstractions

Possible to generate an abstraction with a single-pass read of original graph
AGENDA

- Motivation
- Background: out-of-core processing & graph abstraction
- **System implementation**
  - Case study & benchmarking
  - Conclusion
  - Discussion
WONDERLAND: OVERVIEW

Data model same as Gridgraph:
- Mutable vertex property
- Read-only edges

Two-phase workflow:
- Abstraction generation
- Query processing

1 // Abstraction Generation
2 abstraction = on-disk = {}
3 while not graph.empty():
4   abstraction = {abstract, graph.PopN(...)}
5   abstract, deleted = Select(abstract, X)
6   on-disk = {on-disk, deleted}
7
8 // Query Processing
9 foreach query
10   in_memory_graph = {abstract}
11   while not converge
12     Process(query, in_memory_graph)
13     load = Choose(on-disk)
14   in_memory_graph = {abstract, load}
// Abstraction Generation

Initialize empty abstraction

Expand abstraction with edges from graph

Select set of $X$ edges to remain in abstraction

Edges not selected dumped/streamed to disk

1 // Abstraction Generation
2 abstraction = on-disk = {}
3 while not graph.empty()
4 abstraction = {abstract, graph.PopN(...)}
5 abstract, deleted = Select(abstract, X)
6 on-disk = {on-disk, deleted}

// Query Processing

foreach query

10 in_memory_graph = {abstract}
11 while not converge
12 Process(query, in_memory_graph)
13 load = Choose(on-disk)
14 in_memory_graph = {abstract, load}
// Abstraction Generation
abstraction = on-disk = {}
while not graph.empty():
  abstraction = {abstract, graph.PopN(...)}
  abstract, deleted = Select(abstract, X)
  on-disk = {on-disk, deleted}

// Query Processing
foreach query
  in_memory_graph = {abstract}
  while not converge
    Process(query, in_memory_graph)
    load = Choose(on-disk)
  in_memory_graph = {abstract, load}

Suitable for out-of-core systems!
Only expanded abstract loaded into memory in each iteration; limited edges selected
Original graph read only once
Generated abstraction only has edges from original graph
Initialize empty abstraction
Expand abstraction with edges from graph
Select set of $X$ edges to remain in abstraction
Edges not selected dumped/streamed to disk

// Abstraction Generation
abstraction = on-disk = {}
while not graph.empty()
    abstraction = {abstract, graph.PopN(...)}
    abstract, deleted = Select(abstract, X)
    on-disk = {on-disk, deleted}

// Query Processing
foreach query
    in_memory_graph = {abstract}
    while not converge
        Process(query, in_memory_graph)
        load = Choose(on-disk)
        in_memory_graph = {abstract, load}

Compute on entire graph, or a round
Only store abstracted graph in memory
Compute on a subgraph, or an iteration
Part of on-disk graph chosen to load into memory
New in-memory subgraph created
GridGraph – information propagation is limited per round (each edge processed once), so slower convergence

Wonderland – Abstraction is a “bridge”, and common abstractions are shared among iterations

// Abstraction Generation
abstraction = on-disk = {}
while not graph.empty():
    abstraction = {abstract, graph.PopN(...)}
    abstract, deleted = Select(abstract, X)
    on-disk = {on-disk, deleted}

// Query Processing
foreach query in_memory_graph = {abstract}
while not converge:
    Process(query, in_memory_graph)
    load = Choose(on-disk)
    in_memory_graph = {abstract, load}
// Abstraction Generation
abstraction = on-disk = {}
while not graph.empty():
    abstraction = {abstract, graph.PopN(...)}
    abstract, deleted = Select(abstract, X)
    on-disk = {on-disk, deleted}

// Query Processing
foreach query
    in_memory_graph = {abstract}
    while not converge
        load = Choose(on-disk)
        Process(query, in_memory_graph)
        in_memory_graph = {abstract, load}
How do we select edges for our abstraction that will be suitable for a variety of applications?

How do we reduce the number of random disk accesses?

How do we implement the choose function that decides which part of the on-disk graph we load next?

What is a user-friendly, intuitive way of programming the process function?

```c
// Abstraction Generation
abstraction = on-disk = {}
while not graph.empty()
  abstraction = {abstract, graph.PopN(...)}
  abstract, deleted = Select(abstract, X)
  on-disk = {on-disk, deleted}

// Query Processing
foreach query
  in_memory_graph = {abstract}
  while not converge
    Process(query, in_memory_graph)
    load = Choose(on-disk)
    in_memory_graph = {abstract, load}
```
SELECTING EDGES

What do users need to provide?

1 // Input
2 X = size of abstraction
3 B = the number of edges loaded per iteration
4 S = maximum size of each edge grid
5 W = width of grid

In practice, setting B/X = 0.25 is a reasonable choice. Picking X a bit harder— we'll come back to this.
SELECTING EDGES

What do users need to provide?
What level of abstraction do we give to the user?

1 // Input
2 X = size of abstraction
3 B = the number of edges loaded per iteration
4 S = maximum size of each edge grid
5 W = width of grid

In practice, setting B/X = 0.25 is a reasonable choice. Picking X a bit harder— we’ll come back to this.

Define low-level and high-level APIs for selecting edges:

Low-level API
- Select(vector<Edge>& abstract, size_t X)
- Arbitrary method to sort edges

High-level API
- Edge-priority selection: EdgePriority(Edge e)
- Generate abstraction containing as few weakly connected components as possible with disjoint-set DS— first, join disconnected components, then add edges by priority
How do we store the graph in memory, noting that we process in grids?

Reduce random edge accesses:
- Abstraction edges at header because used in tandem with all other grids
- Typically edges of abstraction are cached

Reduce random vertex accesses:
- Initially, we have a vertex file anyway!
- Possible that non-contiguous vertex IDs are accesses…

| Vertex File | | | | |
|-------------|-------------|-------------|-------------|
| ID^{new}=0  | ...         | ID^{new}=x  | ...         | ID^{new}=|V|-1 |

<table>
<thead>
<tr>
<th>Edge File</th>
<th>Abstract edge grid</th>
<th>Grid 0</th>
<th>...</th>
<th>Grid y</th>
</tr>
</thead>
</table>
How do we store the graph in memory, noting that we process in grids?

Reduce random edge accesses:
- Abstraction edges at header because used in tandem with all other grids
- Typically edges of abstraction are cached

Reduce random vertex accesses:
- Initially, we have a vertex file anyway!
- Possible that non-contiguous vertex IDs are accesses...
- Remap so abstract vertices are at header (but order preserved as much as possible)

<table>
<thead>
<tr>
<th>Vertex File</th>
<th>Edge File</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID_{\text{new}} = 0 ... ID_{\text{new}} = x ... ID_{\text{new}} =</td>
<td>V</td>
</tr>
</tbody>
</table>

Abstract vertices:
- 2, 4

1 2 3 4

Abstract vertices: 2, 4

2 4 1 3
**Reducing Disk Accesses**

How do we store the graph in memory, noting that we process in grids?

How is the remapping done in practice?

- Keep in-memory vector with abstract vertices
- Binary search on this. For a given vertex $X$, we know: 1) is it in the abstraction and 2) the number of vertices $Y$ remaining in the abstract with smaller ID than $X$
  - If it remained, remap to $Y$
  - If it was removed, remap to $X - Y + \text{(number of vertices that remained in abstract)}$

<table>
<thead>
<tr>
<th>Abstract vertices</th>
<th>Other vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID$^{\text{new}=0}$</td>
<td>...</td>
</tr>
</tbody>
</table>

**Vertex File**

**Edge File**

How is the remapping done in practice?

![Diagram of remapping process]

- Abstract vertices: 2, 4
- ID$^{\text{new}}$: 0, 1, 2, 3, 4

Abstract edge grid: Grid 0, ..., Grid $y$
REDUCING DISK ACCESSES

How do we store the graph in memory, noting that we process in grids?

### Vertex File

| ID\text{\text{new}}=0 | ... | ID\text{\text{new}}=x | ... | ID\text{\text{new}}=|V|-1 |

### Edge File

| Abstract edge grid | Grid 0 | ... | Grid y |

How are edges stored?

- If a grid contains $X$ edges and is above the memory limit, partitioned into $\lfloor X/S \rfloor$ edges (either randomly or by priority)
- Edges contiguously stored in CSR format (locating outgoing edges of a vertex in constant time)
- Grids are always loaded as a whole
REDUCING DISK 
ACCESSSES

How do we store the graph in memory, noting that we process in grids?

2

<table>
<thead>
<tr>
<th>Abstract vertices</th>
<th>Other vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID\text{\textsuperscript{new}}=0</td>
<td>...</td>
</tr>
</tbody>
</table>

Vertex File

<table>
<thead>
<tr>
<th>Edge File</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract edge grid</td>
</tr>
</tbody>
</table>

How are edges stored?

- If a grid contains $X$ edges and is above the memory limit, partitioned into $\lfloor X/S \rfloor$ edges (either randomly or by priority)
- Edges contiguously stored in CSR format (locating outgoing edges of a vertex in constant time)
- Grids are always loaded as a whole

How many passes through the edge list is required for grid partitioning?

- Counting edges per grid (1)
- Write edges to locations (1)
- Total: 2 passes
REDUCING DISK ACCESSES

How do we store the graph in memory, noting that we process in grids?

<table>
<thead>
<tr>
<th>Abstract vertices</th>
<th>Other vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID$^{\text{new}} = \emptyset$</td>
<td>...</td>
</tr>
<tr>
<td>ID$^{\text{new}} = x$</td>
<td>...</td>
</tr>
<tr>
<td>ID$^{\text{new}} =</td>
<td>V</td>
</tr>
</tbody>
</table>

Vertex File

<table>
<thead>
<tr>
<th>Edge File</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract edge grid</td>
</tr>
</tbody>
</table>

Edges contiguously stored in CSR format (locating outgoing edges of a vertex in constant time)

How are edges stored?

- If a grid contains $X$ edges and is above the memory limit, partitioned into $\lfloor X/S \rfloor$ edges (either randomly or by priority)
- Edges contiguously stored in CSR format (locating outgoing edges of a vertex in constant time)
- Grids are always loaded as a whole

How many passes through the edge list is required for grid partitioning?

- Counting edges per grid (1)
- Write edges to locations (1)
- Total: 2 passes

Combine first pass with remapping procedure.

**Preprocessing total: 3 passes of edges** (selecting abstraction, remapping, partitioning)
How do we select what edges we load from disk to get our in-memory graph?

Problem: this is very application-specific, and we wanted to develop an application-agnostic framework.

First, enforce users can only choose grid partitions.

Have users specify the order in which they evaluate grids to enforce a scheduling.
How do we actually express an algorithm, or process, to run a query on the graph?

**Low-level API**
- Do-it-yourself
  - Define `Process(Graph& g)` function
  - Given a vector of loaded grids, including abstraction

**High-level API**
- “Think like a vertex”
  - Define function to process on a per-vertex basis
  - Provide multi-threaded executor to parallelize
  - Reuse execution engine already present in literature
How do we actually express an algorithm, or process, to run a query on the graph?

**Low-level API**

\[ \text{Process}(\text{Graph} \& \ g) \]

Given a list of grids (including abstraction, which is a special grid), user has access to:
- List of edges in the grid
- Smallest and largest ID of source vertices
- Access to endpoints for each vertex
EXECUTION OF QUERY

How do we actually express an algorithm, or process, to run a query on the graph?

Low-level API

Process(Graph& g)

Given a list of grids (including abstraction, which is a special grid), user has access to:
• List of edges in the grid
• Smallest and largest ID of source vertices
• Access to endpoints for each vertex

High-level API

func VertexProgram(Graph& g, Index u)
// Iterating loaded edges
foreach grid in g.loaded_grids
  foreach edge in [grid.StartEdge(u), grid.EndEdge(u)]
    ProcessEdge(g.Vertex(u), edge, g.Vertex(edge.destination))

// Update priority of grids
foreach grid in g.all_grids
  if u >= grid.EndVertex(): continue
  if u < grid.StartVertex(): continue
  UpdatePriority(g.Vertex(u), g.Priority(grid))
// Input
X = size of abstraction
B = the number of edges loaded per iteration
S = maximum size of each edge grid
W = width of grid

// Generating Abstract
abstract = vector<Edge>()
on-disk = fstream(...)
while not graph.empty
    abstract = {abstract, graph.PopN(B)}
    abstract, deleted = Select(abstract, X)

// Remapping and Partitioning
abstract, grids = Partition(abstract, on-disk)

// Processing Queries
foreach query
    in_memory_graph = {abstract}
    worklist = BootstrapWorklist(in_memory_graph, query)
    while not converge
        // Worklist-based processing
        while not worklist.empty()
            u = worklist.pop()
            for e in in_memory_graph.loaded_edges(u)
                ProcessEdge(u, e)
            Append worklist accordingly
        Update priority of grids accordingly
    // Bootstrap next iteration
    {grid1, grid2, ...} = Choose(grids, B)
    in_memory_graph = {abstract, grid1, grid2, ...}
    worklist = BootstrapWorklist(in_memory_graph, query)
AGENDA

• Motivation
• Background: out-of-core processing & graph abstraction
• System implementation
• **Case study & benchmarking**
• Conclusion
• Discussion
CASE STUDY: SHORTEST-PATH

- Given a weighted graph with a source (src) and a destination (dst).
- Negative edge weights allowed, but no negative weight cycles.
- Attach a dist property to each vertex and run Dijkstra’s.
- Relaxation: \( \text{dist}[u] + w[u, v] < \text{dist}[v] \)
First, we define the necessary steps for an edge relaxation...

Perform relaxation if necessary
“Activate” edge destination
Push to worklist or process all loaded vertices once

```
func VertexProgram(Graph& g, Index u)
float src_dist = g.Vertex(u).dist

// Iterating loaded edges
foreach grid in g.loaded_grids
    foreach edge in [grid.StartEdge(u), grid.EndEdge(u))
        float new_dist = src_dist + edge.weight
        float& dst_dist = g.Vertex(edge.destination).dist
        if new_dist < dst_dist
            dst_dist = new_dist
            g.Active(edge.destination)
            Worklist.push(edge.destination)

// Update priority of grids
foreach i in [0, g.all_grids.size())
    Grid& grid = g.all_grids[i]
    if u >= grid.EndVertex(): continue
    if u < grid.StartVertex(): continue
    float new_priority = -(src_dist + min_expect[i, dst])
    if grid.priority < new_priority
        grid.priority = new_priority
```
Then, we need to update the priority of the given grid.

Intuition: estimate lower-bound on distance. The smaller this is, the higher the priority.

```csharp
func VertexProgram(Graph& g, Index u)
{
    float src_dist = g.Vertex(u).dist

    // Iterating loaded edges
    foreach grid in g.loaded_grids
    {
        foreach edge in [grid.StartEdge(u), grid.EndEdge(u)]
        {
            float new_dist = src_dist + edge.weight
            float& dst_dist = g.Vertex(edge.destination).dist
            if new_dist < dst_dist
            {
                dst_dist = new_dist
                g.Active(edge.destination)
                workList.push(edge.destination)
            }
        }
    }

    // Update priority of grids
    foreach i in [0, g.all_grids.size())
    {
        Grid& grid = g.all_grids[i]
        if u >= grid.EndVertex(): continue
        if u < grid.StartVertex(): continue
        float new_priority = -(src_dist + min_expect[i, dst])
        if grid.priority < new_priority
        {
            grid.priority = new_priority
        }
    }
}
```
Then, we need to update the priority of the given grid.

Intuition: estimate lower-bound on distance. The smaller this is, the higher the priority.

Define \( \text{min\_expect}[i, \text{dst}] \) for each grid and each \( \text{dst} \) to be the precalculated value of the lower bound of a path starting from an edge in each grid to \( \text{dst} \).

```cpp
func VertexProgram(Graph& g, Index u)
float src_dist = g.Vertex(u).dist

// Iterating loaded edges
foreach grid in g.loaded_grids
  foreach edge in [grid.StartEdge(u), grid.EndEdge(u))
    float new_dist = src_dist + edge.weight
    float& dst_dist = g.Vertex(edge.destination).dist
    if new_dist < dst_dist
      dst_dist = new_dist
      g.Active(edge.destination)
      Worklist.push(edge.destination)

// Update priority of grids
foreach i in [0, g.all_grids.size())
  Grid& grid = g.all_grids[i]
  if u >= grid.EndVertex(): continue
  if u < grid.StartVertex(): continue
  float new_priority = -(src_dist + min_expect[i, dst])
  if grid.priority < new_priority
    grid.priority = new_priority
```
Then, we need to update the priority of the given grid.

Intuition: estimate lower-bound on distance. The smaller this is, the higher the priority.

Define min_expect[i, dst] for each grid and each dst to be the precalculated value of the lower bound of a path starting from an edge in each grid to dst.

Current priority: p
But, if:
Vertex(u).dist + min_expect[i, dst] < p
Update priority with its negation

```cpp
1  func VertexProgram(Graph& g, Index u)
2     float src_dist = g.Vertex(u).dist
3
4     // Iterating loaded edges
5     foreach grid in g.loaded_grids
6         foreach edge in [grid.StartEdge(u), grid.EndEdge(u)]
7             float new_dist = src_dist + edge.weight
8             float& dst_dist = g.Vertex(edge.destination).dist
9             if new_dist < dst_dist
10                 dst_dist = new_dist
11                 g.Active(edge.destination)
12                 Worklist.push(edge.destination)

13     // Update priority of grids
14     foreach i in [0, g.all_grids.size())
15         Grid& grid = g.all_grids[i]
16         if u >= grid.EndVertex(): continue
17         if u < grid.StartVertex(): continue
18         float new_priority = -(src_dist + min_expect[i, dst])
19         if grid.priority < new_priority
20             grid.priority = new_priority
```
Then, we need to update the priority of the given grid.

Intuition: estimate lower-bound on distance. The smaller this is, the higher the priority.

Define min_expect[i, dst] for each grid and each dst to be the precalculated value of the lower bound of a path starting from an edge in each grid to dst.

Computing this could be hard, so instead, we let each grid keep one min_expect, which is the minimum edge weight in the grid.
# Shortest-Path Optimizations

<table>
<thead>
<tr>
<th><strong>Graph abstraction</strong></th>
<th>Edge priority is negative of its weight</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lightest $X$ edges always in abstraction</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Upper-bound</strong></th>
<th>Processing a sequence of SP queries (not SSSP)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Can use current dist priority of dst vertex as upper-bound of relaxation</td>
</tr>
</tbody>
</table>

| **Selecting Loading** | If all edges in grid are not activated, skip over |
EVALUATION ENVIRONMENT

- Two 8-core Intel Xeon CPUs
- Set memory limits per query
- Compare against GridGraph and Galois, as well as out-of-core Galois (named LigraChi-g)
- Average results from 30 query pairs
- Reported results on LiveJournal (790 MB) and Twitter (17 GB) graphs
**SHORTEST-PATH PERFORMANCE: BENCHMARKS**

For high memory limits, devolves into fully in-memory setting, and Galois and Wonderland outperform GridGraph significantly.

For out-of-core system, Wonderland outperforms GridGraph because of abstraction (always keep certain things in memory).
Three sources of speedup:

1. Bootstrapping an initial result
2. Abstraction-enabled information propagation
3. Abstraction-guided priority scheduling

If only (1), speedups are limited as memory limit goes down
Adding (2), extra speedup across the board
Adding (3), extra speedup across the board
But, (1) most important if memory limit is relatively high
SHORTEST PATH CASE STUDY
ABSTRACTION SIZE SENSITIVITY

![Graph](image)
SHORTEST PATH CASE STUDY
MULTI-THREAD SPEEDUP

![Graph showing the relationship between the number of threads and execution time for different memory limits. The graph has a y-axis labeled 'Execution Time (second)' and an x-axis labeled 'Number of Threads'. The legend indicates lines for different memory limits: Limit = 8G, Limit = 2G, Limit = 4G, and Limit = 1G. Each line shows a decrease in execution time as the number of threads increases.](image)
We define the selectivity of the query to denote how much of the graph structure the query necessitates (# total paths)/(# paths we check):

- **Any path queries**: reachability, weakly-connected components (WCC)
- **All path queries**: shortest path (can be pruned from an exhaustive search, so high-selectivity), widest path (maximize weight of min-weight edge)
## SHORTEST PATH CASE STUDY

We define the **selectivity** of the query to denote how much of the graph structure the query necessitates:

- **Any path queries**: reachability, weakly connected components (WCC)
- **All path queries**: shortest path (can be pruned from an exhaustive search, but high selectivity), widest path (maximize weight of min-weight edge)

<table>
<thead>
<tr>
<th>Memory Limit</th>
<th>1/2</th>
<th>1/4</th>
<th>1/8</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Reachbility</strong></td>
<td>LiveJournal</td>
<td>19.8×</td>
<td>16.1×</td>
</tr>
<tr>
<td></td>
<td>Twitter</td>
<td>86.6×</td>
<td>67.0×</td>
</tr>
<tr>
<td></td>
<td>Frindster</td>
<td>376×</td>
<td>247×</td>
</tr>
<tr>
<td></td>
<td>Dimacs</td>
<td>55.3×</td>
<td>39.1×</td>
</tr>
<tr>
<td><strong>WCC</strong></td>
<td>LiveJournal</td>
<td>4.85×</td>
<td>3.75×</td>
</tr>
<tr>
<td></td>
<td>Twitter</td>
<td>6.18×</td>
<td>5.90×</td>
</tr>
<tr>
<td></td>
<td>Frindster</td>
<td>7.11×</td>
<td>6.20×</td>
</tr>
<tr>
<td></td>
<td>Dimacs</td>
<td>9.03×</td>
<td>4.32×</td>
</tr>
<tr>
<td><strong>Shortest Path</strong></td>
<td>LiveJournal</td>
<td>9.92×</td>
<td>8.05×</td>
</tr>
<tr>
<td></td>
<td>Twitter</td>
<td>24.7×</td>
<td>20.9×</td>
</tr>
<tr>
<td></td>
<td>Frindster</td>
<td>36.2×</td>
<td>24.2×</td>
</tr>
<tr>
<td></td>
<td>Dimacs</td>
<td>17.8×</td>
<td>5.91×</td>
</tr>
<tr>
<td><strong>Widest Path</strong></td>
<td>LiveJournal</td>
<td>11.3×</td>
<td>6.18×</td>
</tr>
<tr>
<td></td>
<td>Twitter</td>
<td>9.68×</td>
<td>6.91×</td>
</tr>
<tr>
<td></td>
<td>Frindster</td>
<td>20.1×</td>
<td>17.5×</td>
</tr>
<tr>
<td></td>
<td>Dimacs</td>
<td>63.2×</td>
<td>41.3×</td>
</tr>
</tbody>
</table>
Even for one query, Wonderland is faster than GridGraph.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>GridGraph</th>
<th>Wonderland Random</th>
<th>Wonderland Order</th>
<th>Wonderland Connectivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>LiveJournal</td>
<td>5.53</td>
<td>7.09</td>
<td>9.15</td>
<td>18.6</td>
</tr>
<tr>
<td>Twitter</td>
<td>92.8</td>
<td>101</td>
<td>161</td>
<td>317</td>
</tr>
<tr>
<td>Frindster</td>
<td>235</td>
<td>295</td>
<td>398</td>
<td>912</td>
</tr>
<tr>
<td>Dimacs</td>
<td>3.79</td>
<td>6.29</td>
<td>6.35</td>
<td>11.6</td>
</tr>
</tbody>
</table>
SCOPE OF APPLICATION

• Higher speedup if problem has a higher selectivity (BFS, MST)
• Basic graph operations; could be extended to clustering, matching, flow, etc.
• Wonderland can also be faster in computing sparse matrix-vector multiplication algs like PageRank, but due to Galois engine, not abstraction
AGENDA

- Motivation
- Background: out-of-core processing & graph abstraction
- System implementation
- Case study & benchmarking
- Conclusion
- Discussion
CONCLUSION

- Wonderland: novel, out-of-core graph processing framework
  - Extract effective abstractions from original graph
  - Use this to enable effective information propagation
  - Use this to enable priority scheduling for faster convergence

- Drastic speedup over other state-of-the-art systems
DISCUSSION

• More information on tuning parameters during benchmarking
• Useful to see piecewise breakdown, but unclear how bootstrapping of initial result happens
• Theoretical work into deriving bounds on the best \( X \)
• Paper laid out well but should have done a more thorough review of literature—many of the concepts (besides abstraction) are not novel to this paper in particular