A Pattern-Aware Graph Mining System

Kasra Jamshidi  Rakesh Mahadasa  Keval Vora

Simon Fraser University

https://github.com/pdclab/peregrine
Why should you pay attention?
Peregrine executes 700x faster
Peregrine consumes 100x less memory
Peregrine scales to 100x larger datasets
On 8x fewer machines
With a more expressive API
Graph Mining

Data Graph
Graph Mining

Data Graph
Graph Mining

Data Graph

Subgraph
Graph Mining

Data Graph

Pattern
Graph Mining

Edge-Induced
Graph Mining

Edge-Induced
Graph Mining

Edge-Induced
Graph Mining

Vertex-Induced
Graph Mining

Vertex-Induced
Graph Mining

Vertex-Induced
Graph Mining

Data Graph
Graph Mining

Data Graph

Vertex-Induced
Graph Mining

Data Graph

Vertex-Induced

Edge-Induced
Graph Mining

Data Graph

Frequent Patterns (Edge-Induced)
Graph Mining

Data Graph

Unlabeled Pattern Distribution (Vertex-Induced)
Scalability Challenge
Scalability Challenge

- 4-motif counting on Orkut graph ($|V| = 13M$, $|E| = 117M$)

![Motif Diagrams]
Scalability Challenge

- 4-motif counting on Orkut graph (|V| = 13M, |E| = 117M)

123,503,340,341,270 subgraphs
System Requirements
System Requirements

Uniqueness
System Requirements

Uniqueness

\[ \begin{align*}
  x & \quad y \\
  z & \quad \text{Uniqueness} \\
  x & \quad y \\
  y & \quad z \\
  z & \quad x
\end{align*} \]
System Requirements

Uniqueness

- z
- y
- x

- x
- y
- z

- y
- z
- x

- x
- y
- z

28
System Requirements

Uniqueness

Structure
System Requirements

Uniqueness

Structure
System Requirements

Uniqueness

Structure

Interestingness
System Requirements

Uniqueness
Structure
Interestingness
System Requirements

Uniqueness

Structure

Interestingness
System Requirements

Uniqueness
Structure
Interestingness
Existing Work

Uniqueness
Arabesque (SOSP ‘15)
RStream (OSDI ‘18)

Structure
Fractal (SIGMOD ‘19)

Interestingness
AutoMine (SOSP ‘19)
Existing Work

Uniqueness
Arabesque (SOSP ‘15)
RStream (OSDI ‘18)

Structure
Fractal (SIGMOD ‘19)

Interestingness
AutoMine (SOSP ‘19)

Overlook user requirements
Existing Work

Uniqueness
Arabesque (SOSP ‘15)
RStream (OSDI ‘18)
Fractal (SIGMOD ‘19)
AutoMine (SOSP ‘19)

Structure

Interestingness

Overlook user requirements
Per-subgraph computations
Pattern Awareness

Pattern Selection

Pattern Matching
Pattern Awareness

Pattern Selection

Pattern Matching
Pattern Awareness

Pattern Selection
#include "Peregrine.hh"
using namespace Peregrine;

void motifCounting(int size)
{
    DataGraph G("path/to/graph/");
    auto patterns = PatternGenerator::all(size, VERTEX_INDUCED);
    auto counts = count(G, patterns);

    for (auto &[pattern, n] : counts)
        std::cout << pattern << "  " << n << std::endl;
}
#include "Peregrine.hh"
using namespace Peregrine;

void motifCounting(int size) {
    DataGraph G("path/to/graph/");
    auto patterns = PatternGenerator::all(size, VERTEX_INDUCED);
    auto counts = count(G, patterns);

    for (auto &[pattern, n] : counts)
        std::cout << pattern << " " << n << std::endl;
}
Pattern Programming

```cpp
#include “Peregrine.hh”
using namespace Peregrine;

void motifCounting(int size)
{
    DataGraph G(“path/to/graph/”);
    auto patterns = PatternGenerator::all(size, VERTEX_INDUCED);
    auto counts = count(G, patterns);

    for (auto &[pattern, n] : counts)
        std::cout << pattern << “ ” << n << std::endl;
}
```
Pattern Programming

#include "Peregrine.hh"
using namespace Peregrine;

void motifCounting(int size)
{
    DataGraph G("path/to/graph/");
    auto patterns = PatternGenerator::all(size, VERTEX_INDUCED);
    auto counts = count(G, patterns);

    for (auto &[pattern, n] : counts)
    {
        std::cout << pattern << " " << n << std::endl;
    }
}
DataGraph G("path/to/graph/");
auto patterns = PatternGenerator::all(size, VERTEX_INDUCED);

auto counts = count(G, patterns);
DataGraph G("path/to/graph/");
auto patterns = PatternGenerator::all(size, VERTEX_INDUCED);
patterns[0].set_labels({‘a’, ‘b’, ‘c’, ‘d’});

auto counts = count(G, patterns);
Pattern Programming

```cpp
DataGraph G(“path/to/graph/”);
auto patterns = PatternGenerator::all(size, VERTEX_INDUCED);
patterns[0].set_labels(’a’, ’b’, ’c’, ’d’);
patterns[0].add_edge(1, 5);

auto counts = count(G, patterns);
```
DataGraph G("path/to/graph/");
auto patterns = PatternGenerator::all(size, VERTEX_INDUCED);
patterns[0].set_labels({'a', 'b', 'c', 'd'});
patterns[0].add_edge(1, 5);
patterns.emplace_back("path/to/pattern.txt");
auto counts = count(G, patterns);
DataGraph G("path/to/graph/");
auto patterns = PatternGenerator::all(size, VERTEX_INDUCED);
auto pattern = Pattern().add_edge(1, 2)
    .add_edge(1, 3)
    .add_edge(2, 3);
auto counts = count(G, {pattern});
#include "Peregrine.hh"
using namespace Peregrine;

void motifCounting(int size)
{
    DataGraph G("path/to/graph/");
    auto patterns = PatternGenerator::all(size, VERTEX_INDUCED);
    auto counts = count(G, patterns);

    for (auto &[pattern, n] : counts)
        std::cout << pattern << " " << n << std::endl;
}
#include "Peregrine.hh"
using namespace Peregrine;

void motifCounting(int size) {
    DataGraph G("path/to/graph/");
    auto patterns = PatternGenerator::all(size, VERTEX_INDUCED);
    auto counts = count(G, patterns);

    for (auto &[pattern, n] : counts) {
        std::cout << pattern << " " << n << std::endl;
    }
}
void frequentSubgraphMining()
{
    DataGraph G("path/to/graph/");
    auto patterns = PatternGenerator::all(2, EDGE_INDUCED);
    auto mapDomain = [](auto &&match, auto &&aggregator)
    {
        aggregator.map(match.pattern, match.mapping);
    };

    auto results = match<Pattern, Domain>(G, patterns, mapDomain);

    for (auto &[pattern, frequency] : results)
        std::cout << pattern << " " << frequency << std::endl;
}
void frequentSubgraphMining()
{
    DataGraph G("path/to/graph/");
    auto patterns = PatternGenerator::all(2, EDGE_INUEDC);
    auto mapDomain = [] (auto &&match, auto &&aggregator)
    { aggregator.map(map.match.pattern, match.mapping); };

    auto results = match<Pattern, Domain>(G, patterns, mapDomain);

    for (auto &[pattern, frequency] : results)
        std::cout << pattern << " " << frequency << std::endl;
}
void frequentSubgraphMining()
{
    DataGraph G("path/to/graph/");
    auto patterns = PatternGenerator::all(2, EDGE_INDUCED);
    auto mapDomain = [](auto &&match, auto &&aggregator)
    {
        aggregator.map(match.pattern, match.mapping);
    };

    auto results = match<Pattern, Domain>(G, patterns, mapDomain);

    for (auto &[pattern, frequency] : results)
    {
        std::cout << pattern << " " << frequency << std::endl;
    }
}
void frequentSubgraphMining()
{
    DataGraph G("path/to/graph/");
    auto patterns = PatternGenerator::all(2, EDGE_INDUCED);
    auto mapDomain = [](auto &&match, auto &&aggregator)
    {
        aggregator.map(map(match.pattern, match.mapping));
    };

    auto results = match<Pattern, Domain>(G, patterns, mapDomain);

    for (auto &[pattern, frequency] : results)
        std::cout << pattern << " " << frequency << std::endl;
}
Pattern Awareness

Pattern Selection

Pattern Matching
Pattern Awareness

Pattern Selection

Anti-Edge
Pattern Awareness

Pattern Selection

Anti-Vertex
Pattern Awareness

Pattern Selection

Pattern Matching
Pattern Awareness

Pattern Selection

Pattern Matching
Pattern Awareness

- Symmetry breaking (RECOMB ‘07)
- Core pattern reduction (SIGMOD ‘16)
Symmetry Breaking
Symmetry Breaking
Symmetry Breaking
Symmetry Breaking
Symmetry Breaking

\[ u \quad x \quad y \quad v \]
Symmetry Breaking
Symmetry Breaking

Data Graph

Worker 1
Symmetry Breaking

Data Graph

Worker 1
Symmetry Breaking

Data Graph

Worker 1
Symmetry Breaking

Data Graph

Worker 2

Worker 1
Symmetry Breaking

Data Graph

Worker 2

Worker 1
Symmetry Breaking

Data Graph

Worker 2

Worker 1
Symmetry Breaking

Data Graph

Worker 2

Worker 1
Symmetry Breaking

Data Graph

Worker 2

Worker 1
Symmetry Breaking

Data Graph

Worker 2

Worker 1
Symmetry Breaking

\[ u < v \]
Symmetry Breaking

\[ x < y \]
Symmetry Breaking

\[ u < v \]
\[ x < y \]
Core Pattern
Core Pattern

\[
\begin{align*}
    u & \rightarrow x \\
    y & \rightarrow v
\end{align*}
\]
Core Pattern
Core Pattern

Data Graph

Pattern-Unaware
Core Pattern

Data Graph

Pattern-Unaware
Core Pattern

Data Graph

Pattern-Unaware
Core Pattern

Data Graph

Pattern-Unaware
Core Pattern

Data Graph

Pattern-Unaware
Core Pattern

Data Graph

Pattern-Aware
Core Pattern

Data Graph

Pattern-Aware
Core Pattern

Data Graph

Pattern-Aware
Core Pattern

Data Graph

Pattern-Aware
Core Pattern

Data Graph

Pattern-Aware
Core Pattern

Data Graph

Pattern-Aware
Core Pattern

Data Graph

Pattern-Aware
Pattern Awareness

- Symmetry breaking (RECOMB ‘07)
- Core pattern reduction (SIGMOD ‘16)
Pattern Awareness

- Early termination
bool globalClusteringCoefficient(int bound) {
    DataGraph G("path/to/graph/");
    auto triplet = PatternGenerator::star(3);
    int numTriplets = count(G, {triplet});
    auto countAndCheck = [=](auto &&match, auto &&aggregator) {
        int numTriangles = aggregator.readValue(match.pattern);
        if (3*numTriangles/numTriplets > bound) aggregator.stop();
        else aggregator.map(match.pattern, 1);
    };
    auto triangle = PatternGenerator::clique(3);
    auto result = match<Pattern, int>(G, triangle, countAndCheck);
    return 3*result[triangle]/numTriplets > bound;
}
bool globalClusteringCoefficient(int bound)
{
    DataGraph G("path/to/graph/");
    auto triplet = PatternGenerator::star(3);
    int numTriplets = count(G, {triplet});
    auto countAndCheck = [=](auto &&match, auto &&aggregator)
    {
        int numTriangles = aggregator.readValue(match.pattern);
        if (3*numTriangles/numTriplets > bound) aggregator.stop();
        else aggregator.map(match.pattern, 1);
    }
    auto triangle = PatternGenerator::clique(3);
    auto result = match<Pattern, int>(G, triangle, countAndCheck);
    return 3*result[triangle]/numTriplets > bound;
}
bool globalClusteringCoefficient(int bound) {
    DataGraph G("path/to/graph/");
    auto triplet = PatternGenerator::star(3);
    int numTriplets = count(G, {triplet});
    auto countAndCheck = [=](auto &&match, auto &&aggregator) {
        int numTriangles = aggregator.readValue(match.pattern);
        if (3 * numTriangles / numTriplets > bound) aggregator.stop();
        else aggregator.map(match.pattern, 1);
    }
    auto triangle = PatternGenerator::clique(3);
    auto result = match<Pattern, int>(G, triangle, countAndCheck);
    return 3 * result[triangle] / numTriplets > bound;
}
bool globalClusteringCoefficient(int bound) {
    DataGraph G("path/to/graph/");
    auto triplet = PatternGenerator::star(3);
    int numTriplets = count(G, {triplet});
    auto countAndCheck = [=](auto &&match, auto &&aggregator) {
        int numTriangles = aggregator.readValue(match.pattern);
        if (3*numTriangles/numTriplets > bound) aggregator.stop();
        else aggregator.map(match.pattern, 1);
    };
    auto triangle = PatternGenerator::clique(3);
    auto result = match<Pattern, int>(G, triangle, countAndCheck);
    return 3*result[triangle]/numTriplets > bound;
}
bool globalClusteringCoefficient(int bound)
{
    DataGraph G("path/to/graph/");
    auto triplet = PatternGenerator::star(3);
    int numTriplets = count(G, {triplet});
    auto countAndCheck = [=](auto &&match, auto &&aggregator)
    {
        int numTriangles = aggregator.readValue(match.pattern);
        if (3*numTriangles/numTriplets > bound) aggregator.stop();
        else aggregator.map(match.pattern, 1);
    };
    auto triangle = PatternGenerator::clique(3);
    auto result = match<Pattern, int>(G, triangle, countAndCheck);
    return 3*result[triangle]/numTriplets > bound;
}
bool globalClusteringCoefficient(int bound) {
    DataGraph G(“path/to/graph/”);
    auto triplet = PatternGenerator::star(3);
    int numTriplets = count(G, {triplet});
    auto countAndCheck = [=](auto &&match, auto &&aggregator) {
        int numTriangles = aggregator.readValue(match.pattern);
        if (3*numTriangles/numTriplets > bound) aggregator.stop();
        else aggregator.map(match.pattern, 1);
    }
    auto triangle = PatternGenerator::clique(3);
    auto result = match<Pattern, int>(G, triangle, countAndCheck);
    return 3*result[triangle]/numTriplets > bound;
}
bool globalClusteringCoefficient(int bound) {
    DataGraph G("path/to/graph/");
    auto triplet = PatternGenerator::star(3);
    int numTriplets = count(G, {triplet});
    auto countAndCheck = [=](auto &&match, auto &&aggregator) {
        int numTriangles = aggregator.readValue(match.pattern);
        if (3*numTriangles/numTriplets > bound) aggregator.stop();
        else aggregator.map(match.pattern, 1);
    }
    auto triangle = PatternGenerator::clique(3);
    auto result = match<Pattern, int>(G, triangle, countAndCheck);
    return 3*result[triangle]/numTriplets > bound;
}
bool globalClusteringCoefficient(int bound) {
    DataGraph G(“path/to/graph/”);
    auto triplet = PatternGenerator::star(3);
    int numTriplets = count(G, {triplet});
    auto countAndCheck = [=](auto &&match, auto &&aggregator)
    {
        int numTriangles = aggregator.readValue(match.pattern);
        if (3*numTriangles/numTriplets > bound) aggregator.stop();
        else aggregator.map(match.pattern, 1);
    }
    auto triangle = PatternGenerator::clique(3);
    auto result = match<Pattern, int>(G, triangle, countAndCheck);
    return 3*result[triangle]/numTriplets > bound;
}
bool globalClusteringCoefficient(int bound) {
    DataGraph G("path/to/graph/");
    auto triplet = PatternGenerator::star(3);
    int numTriplets = count(G, {triplet});
    auto countAndCheck = [=](auto &&match, auto &&aggregator) {
        int numTriangles = aggregator.readValue(match.pattern);
        if (3*numTriangles/numTriplets > bound) aggregator.stop();
        else aggregator.map(match.pattern, 1);
    };
    auto triangle = PatternGenerator::clique(3);
    auto result = match<Pattern, int>(G, triangle, countAndCheck);
    return 3*result[triangle]/numTriplets > bound;
}
bool globalClusteringCoefficient(int bound) {
    DataGraph G("path/to/graph/");
    auto triplet = PatternGenerator::star(3);
    int numTriplets = count(G, {triplet});
    auto countAndCheck = [=](auto &&match, auto &&aggregator) {
        int numTriangles = aggregator.readValue(match.pattern);
        if (3*numTriangles/numTriplets > bound) aggregator.stop();
        else aggregator.map(match.pattern, 1);
    };
    auto triangle = PatternGenerator::clique(3);
    auto result = match<Pattern, int>(G, triangle, countAndCheck);
    return 3*result[triangle]/numTriplets > bound;
}
bool globalClusteringCoefficient(int bound)
{
    DataGraph G("path/to/graph/");
    auto triplet = PatternGenerator::star(3);
    int numTriplets = count(G, {triplet});
    auto countAndCheck = [=](auto &&match, auto &&aggregator)
    {
        int numTriangles = aggregator.readValue(match.pattern);
        if (3*numTriangles/numTriplets > bound) aggregator.stop();
        else aggregator.map(match.pattern, 1);
    }
    auto triangle = PatternGenerator::clique(3);
    auto result = match<Pattern, int>(G, triangle, countAndCheck);
    return 3*result[triangle]/numTriplets > bound;
}
Comparison with Existing Work

• Peregrine with 16 logical cores and 32GB RAM
• Arabesque & Fractal with 8x16 logical cores and 8x32GB RAM
• RStream with 96 logical cores and 192GB RAM
Comparison with Existing Work

- Fractal [SIGMOD ’19]: Peregrine 737x, Other System 10^{11.5}
- RStream [OSDI ’18]: Peregrine 866x, Other System 10^{12.5}
- Arabesque [SOSP ’15]: Peregrine 1317x, Other System 10^{13.5}
## Comparison with Existing Work

<table>
<thead>
<tr>
<th>Method</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fractal [SIGMOD ’19]</td>
<td>Failed 22 out of 43 experiments</td>
</tr>
<tr>
<td>RStream [OSDI ’18]</td>
<td>Failed 13 out of 26 experiments</td>
</tr>
<tr>
<td>Arabesque [SOSP ’15]</td>
<td>Failed 14 out of 26 experiments</td>
</tr>
</tbody>
</table>
Effects of Pattern Awareness

Data Graph

- Orkut: Did Not Finish
- US Patents: 11x
- MiCo: 42x

4-Motif Counting Execution Time (s)

Legend:
- Aware
- Unaware
Scalability

![Graph showing scalability with speedup on the y-axis and number of threads on the x-axis. The graph compares ideal scalability (dashed line) and Peregrine scalability (solid line). At 64 threads, ideal speedup is 46x, and Peregrine speedup is 57x. At 96 threads, Peregrine speedup is 57x.](image-url)
Scalability

![Scalability Chart]

- **X-axis:** Number of Threads
- **Y-axis:** Speedup
- **Legend:**
  - Red dashed line: Ideal
  - Blue solid line: Peregrine

Key Points:
- At 64 threads, the ideal speedup is 46x.
- Peregrine achieves a speedup of 57x at 96 threads.
Scalability

[Graph showing scalability with respect to the number of threads. The x-axis represents the number of threads, ranging from 1 to 96. The y-axis represents speedup, ranging from 1 to 60. Two lines are plotted: one dashed line representing the ideal scalability and one solid line representing Peregrine's performance. The dashed line shows an ideal speedup of 57x at 96 threads.]
Scalability

![Graph showing scalability with number of threads and speedup. The graph compares ideal and Peregrine scenarios, with marked speedup at 41x for 48 threads and 46x for 64 threads, reaching 57x at 96 threads.](image-url)
Scalability

The graph represents the scalability of two systems: Ideal and Peregrine. The x-axis shows the number of threads, ranging from 1 to 96, and the y-axis represents speedup, ranging from 1 to 60.

- The Ideal system shows a straight line going through the points (48, 41x) and (96, 57x), indicating linear scalability.
- The Peregrine system also shows linear scalability, with points (48, 46x) and (96, 57x).
Scalability

The graph illustrates the scalability of different systems as the number of threads increases. The red dashed line represents the ideal scalability, while the blue solid line shows the performance of Peregrine. The blue dashed line indicates Peregrine with Hyper-Threading (HT). The graph shows that Peregrine performs well, approaching the ideal scalability, with significant speedups at 41x, 46x, and 57x as the number of threads increases from 1 to 96.
Scalability
Pattern-aware programming and processing models
Pattern-aware programming and processing models

• Shift abstraction from subgraph to pattern
Pattern-aware programming and processing models

• Shift abstraction from subgraph to pattern
• User program is transparent to the system
Pattern-aware programming and processing models

• Shift abstraction from subgraph to pattern
• User program is transparent to the system
• Up to 42x faster than pattern-unaware
• Up to 737x faster than state-of-the-art
Pattern-aware programming and processing models

• Shift abstraction from subgraph to pattern
• User program is transparent to the system
• Up to 42x faster than pattern-unaware
• Up to 737x faster than state-of-the-art

https://github.com/pdclab/peregrine