FOSDEM '23
Rust Devroom

Scalable graph algorithms in Rust (and Python)
Who?

Martin Junghanns
Senior Software Engineer
Graph Data Science at Neo4j
@s1ck@hachyderm.io

Paul Horn
Senior Software Engineer
Graph Data Science at Neo4j
@knutwalker@hachyderm.io
Our day job …

• Neo4j Graph Data Science
  ◦ Plugin for Neo4j Graph Database
  ◦ Provides a collection of graph and machine learning algorithms
  ◦ Customer use cases with up to 10+ bn nodes and 65+ bn edges
  ◦ Top 3 use cases: fraud detection, recommendation, identity resolution

• Product: https://neo4j.com/product/graph-data-science/

• Source: https://github.com/neo4j/graph-data-science/

• Docs: https://neo4j.com/docs/graph-data-science/current/
**Pathfinding & Search**
- A* Shortest Path
- All Pairs Shortest Path
- Breadth & Depth First Search
- Delta-Stepping Single-Source
- Dijkstra Single-Source
- Dijkstra Source-Target
- Minimum Spanning Tree & K-Spanning Tree
- Random Walk
- Yen’s K Shortest Path
- Minimum Directed Steiner Tree

**Centrality & Importance**
- ArticleRank
- Betweenness Centrality & Approx.
- Closeness Centrality
- Degree Centrality
- Eigenvector Centrality
- Harmonic Centrality
- Hyperlink Induced Topic Search (HITS)
- Influence Maximization (CELF)
- PageRank
- Personalized PageRank

**Community Detection**
- Conductance Metric
- K-1 Coloring
- K-Means Clustering
- Label Propagation
- Leiden Algorithm
- Local Clustering Coefficient
- Louvain Algorithm
- Max K-Cut
- Modularity Optimization
- Speaker Listener Label Propagation
- Strongly Connected Components
- Triangle Count
- Weakly Connected Components

**Supervised Machine Learning**
- Link Prediction Pipelines
- Node Classification Pipelines
- Node Regression Pipelines

**Heuristic Link Prediction**
- Adamic Adar
- Common Neighbors
- Preferential Attachment
- Resource Allocations
- Same Community
- Total Neighbors

**Similarity**
- K-Nearest Neighbors (KNN)
- Node Similarity
- Filtered KNN & Node Similarity
- Cosine & Pearson Similarity Functions
- Euclidean Distance Similarity Function
- Euclidean Similarity Function
- Jaccard & Overlap Similarity Functions

**Graph Embeddings**
- Fast Random Projection (FastRP)
- FastRP with Property Weights
- GraphSAGE
- Node2Vec
- HashGNN (Knowledge Graph Embedding)

... and more!
Graph Algorithms in Rust … Why?

- Rust is a popular systems programming language known for its memory safety, modern type system, and native performance
- We are curious, performance-focused engineers who always want to learn more about what’s happening outside of our (JVM) box
- We like Rust and wanted to explore how a graph library for graphs with billion+ nodes and relationships would look like and perform in Rust
The graph project
The graph project

• Started in May 2021 as an experiment / hobby project
  ◦ Pure interest in combining Rust and graph algorithms
  ◦ Initial goal was to learn what level of performance we can achieve
  ◦ Using parallel implementations wherever possible
  ◦ Added more algorithms, features and API improvements over time

• Code is available on GitHub: https://github.com/s1ck/graph
The graph project - crates

- graph_mate**
- server
- app
- graph*
- graph_builder*

* available on crates.io
** available on pypi.org
The graph project - crates - graph_builder

- Graph API for building directed and undirected property graphs

```rust
let g: UndirectedCsrGraph<u64> = GraphBuilder::new()
    .edges([
        (0, 1),
        (0, 2),
        (1, 2),
        (1, 3),
        (2, 4),
        (3, 4)
    ])
    .build();

assert_eq!(g.degree(1), 3);
assert_eq!(g.neighbors(2).as_slice(), &[0, 1, 4]);
assert_eq!(g.neighbors(4).as_slice(), &[2, 3]);
```
The graph project - crates - graph_builder

- Graph API for building directed and undirected property graphs

```rust
let g: DirectedCsrGraph<u64> = GraphBuilder::new()
    .edges([
      (0, 1),
      (0, 2),
      (1, 2),
      (1, 3),
      (2, 4),
      (3, 4)
    ])
    .build();

assert_eq!(g.out_degree(1), 2);
assert_eq!(g.out_neighbors(2).as_slice(), &[4]);
assert_eq!(g.in_neighbors(4).as_slice(), &[2, 3]);
```
The graph project - crates - graph_builder

- Graph API for building directed and undirected property graphs

```rust
let g: DirectedCsrGraph<u64, u32> = GraphBuilder::new()
    .edges([ (0, 1),
              (0, 2),
              (1, 2),
              (1, 3),
              (2, 4),
              (3, 4) ]) .node_values([1, 3, 3, 7, 42]) .build();

assert_eq!(g.node_value(0), 1);
assert_eq!(g.node_value(4), 42);
```
The graph project - crates - graph_builder

- Graph API for building directed and undirected property graphs

```rust
let g: DirectedCsrGraph<u64, u32, f32> = GraphBuilder::new()
    .edges([
    (0, 1, 0.1),
    (0, 2, 0.2),
    (1, 2, 0.3),
    (1, 3, 0.4),
    (2, 4, 0.5),
    (3, 4, 0.6)
    ])
    .node_values([1, 3, 3, 7, 42])
    .build();
```

```rust
assert_eq!(
    g.out_neighbors_with_values(2).as_slice(),
    &[Target::new(4, 0.5)]
);
```
The graph project - crates - graph_builder

- Graph API for building directed and undirected property graphs

```rust
let g: DirectedCsrGraph<u64, u32, f32> = GraphBuilder::new()
    .gdl_str::<u64, _>("(n0 {p: 1}), (n1 {p: 3}),
    (n2 {p: 3}), (n3 {p: 7}),
    (n4 {p: 42}),
    (n0)-[{} f: 0.1 ]->(n1),
    (n0)-[{} f: 0.2 ]->(n2),
    (n1)-[{} f: 0.3 ]->(n2),
    (n1)-[{} f: 0.4 ]->(n3),
    (n2)-[{} f: 0.5 ]->(n4),
    (n3)-[{} f: 0.6 ]->(n4)",
).build()
    .unwrap();
```
The graph project - crates - graph_builder

- Graphs can be created **programmatically** as shown before

- Graphs can be created **from files** using GraphInput implementations
  - **EdgeList** - text file containing ”source target [value]” tuples per line
  - **Graph500** - binary file storing the output of the Graph500\(^1\) data generator
  - **Serialized** - binary file serialized using the graph_builder crate

- Graph creation is fully parallelized using the rayon crate\(^2\)

\(^1\) https://graph500.org
\(^2\) https://crates.io/crates/rayon
The graph project - crates - graph

• Provides a small set of parallel graph algorithms
  ◦ Page Rank
  ◦ Weakly Connected Components
  ◦ Global Triangle Count
  ◦ Single-Source Shortest Path

• Graph algorithms are also parallelized using the rayon crate

• Contributions are very welcome! 🦀
The graph project - crates - graph

```rust
let gdl = "(n0)-->(n1)-->(n2),(n0)-->(n2),(n1)-->(n3)-->(n4),(n2)-->(n4)";

let graph: DirectedCsrGraph<u32> = GraphBuilder::new()
    .csr_layout(CsrLayout::Sorted)
    .gdl_str::<u32, _>(gdl)
    .build()
    .unwrap();

let (scores, _, _) = page_rank(&graph, PageRankConfig::default());

let expected: Vec<f32> = vec![
    0.029999996,
    0.042749994,
    0.06091874,
    0.04816874,
    0.122724354,
];

assert_eq!(scores, expected);
```
The graph project - crates - graph_mate

- Python bindings for the `graph_builder` and the `graph` crates
  - Expose pythonic API for Rust implementation
- Memory management and parallelism done in Rust
- Integrates with `numpy` and `pandas`
- Alpha state, not everything is available yet
- Available on PyPI: `pip install graph-mate`
Demo scenario

- Uses Graph500\(^1\) dataset scale 24 (~17M nodes, ~260M edges)
  - Generates a graph where degrees follow a power-law distribution

- Demo workflow:
  1. Create directed graph from Graph500 binary graph file
  2. Compute Page Rank
  3. Compute Weakly Connected Components
  4. Convert graph to an undirected and relabeled graph
  5. Compute Triangle Count

- 3 Implementations: \texttt{graph_mate}, graph, pyarrow + server

\(^1\) https://graph500.org/?page_id=12#sec-3
Demo 1: graph_mate (Python)

Demo 2: graph (Rust)

Demo 3: pyarrow + server
Lessons Learned
Lessons Learned from building the graph project

• Using Rust as a Java developer (with some understanding of the JVM)
  ◦ Rust paradigms require a different thinking about how to design code
  ◦ Mechanical sympathy improves when working with Rust
  ◦ Different, but nicely integrated ecosystem (Cargo, rust-analyzer, …)
  ◦ Debugging and profiling requires learning about tools from the C/C++ world

• What about the performance?
  ◦ For algorithms that we implemented in Java and Rust, we could see a better performance in Rust for all cases
  ◦ Predictable runtime behaviour
    • No latency spikes, consistent allocation rate
    • AOT compiler and LLVM backend
Outlook
Outlook

• What we want to work on next
  ◦ Add more algorithms
  ◦ Expand the Python and Arrow Server APIs
  ◦ Add algorithm framework to allow users to write their own algos
  ◦ Explore native capabilities even further (SIMD, GPU, …)

• The library is usable, but not battle tested

• What we need from you
  ◦ Feedback (reporting issues, etc.)
  ◦ Contributions!

• For a longer version of this talk with all demos check out YT
Thank you!

Q&A offline