Hewlett Packard Enterprise

Malicious website detection with large scale belief propagation on Spark

Mijung Kim, Jun Li, Manish Marwah, Alexander Ulanov, Carlos Zubieta

Hewlett Packard Labs

Outline

– Introduction

– (Loopy) Belief Propagation

– Implementation on Spark

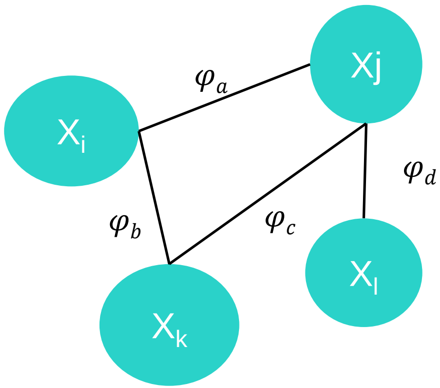
– Experiments

– Conclusions and Future Work

Introduction

Probabilistic Graphical Models

- Combine graphs and probability theory
- Vertices: random variables
- Edges: relationships between variables
- Model joint probability distribution
- Used for probabilistic reasoning

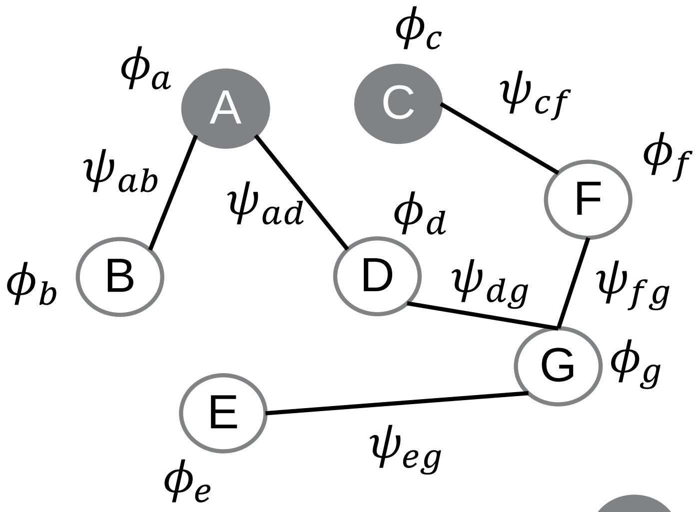


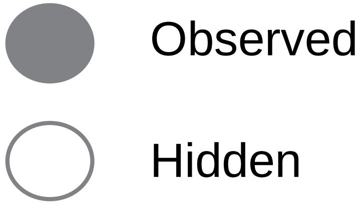
Graphical Model: An Example

Markov Random Field Model

– Model parameters
  – Priors / node factors
  – Edge factors

– Queries
  – P(B)
  – P(B|A, C)
  – P(E|C)
  – ….



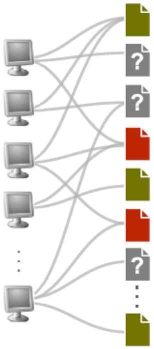


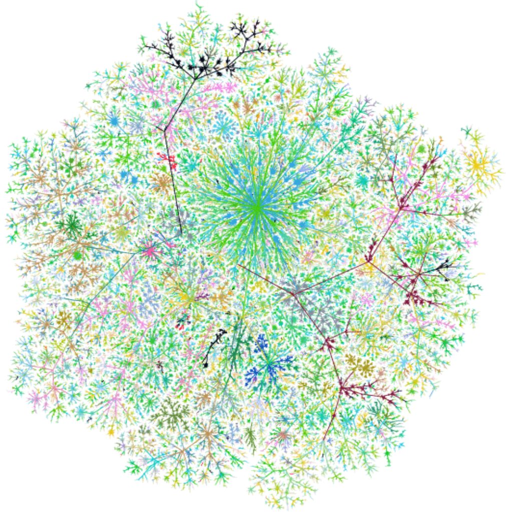
Joint Probability

$$P(\mathcal{X}) = \frac{1}{Z}\prod_{i\in\mathcal{C}}\psi_i$$

Applications

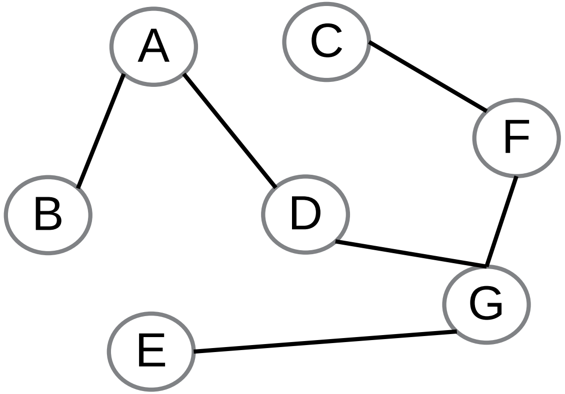
- Computer vision
- Malware detection
- Fraud detection
- Bioinformatics
- Recommender system
- ….





Inference in Graphical Models

- Queries of the graphical model
  - P(A | B, C) ?
  - P (D) ?
  - ***Inference***
- Methods
  - Exact
  - Approximate
    - Variational methods
    - **Loopy Belief Propagation**
    - Sampling based (e.g., Gibbs sampling)



Loopy Belief Propagation Algorithm

- Message passing iterative algorithm
- Exact on trees, approximate on graphs with loops

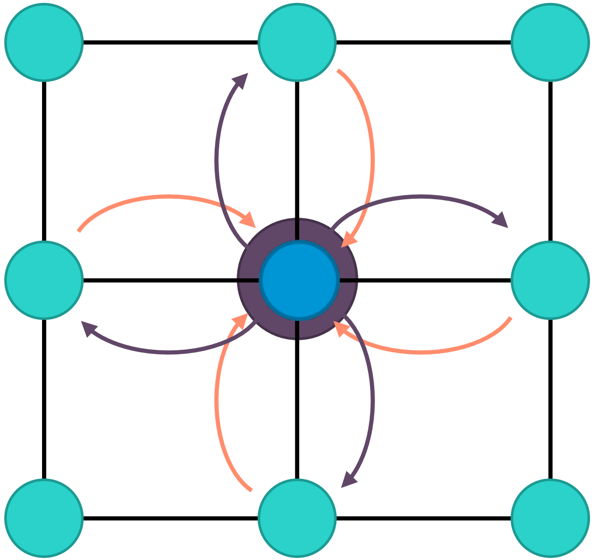
- Initialize messages
- At each node
    - **Read** messages from neighbors
    - **Update** own marginal probability (belief)

$$b(x_i) = \phi(x_i) \prod_{j \in \mathcal{N}(i)} m_{j \to i}$$

    - **Send** updated messages to all neighbors

$$m_{i \to j}(x_j) = \sum_{x_i} \phi(x_i) \psi_{ij}(x_i, x_j) \prod_{k \in \mathcal{N}(i) \backslash j} m_{k \to i}$$

- Repeat until convergence



Application: Detection/Ranking of Malicious Websites

- Goal: To infer maliciousness score of a webhost/webpage

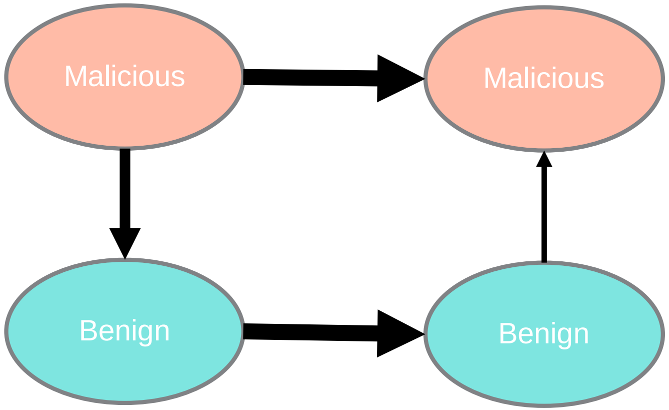
- Given web graph, exploit principle of “homophily”

- Malicious site → Malicious site
- Benign site → Benign site

–

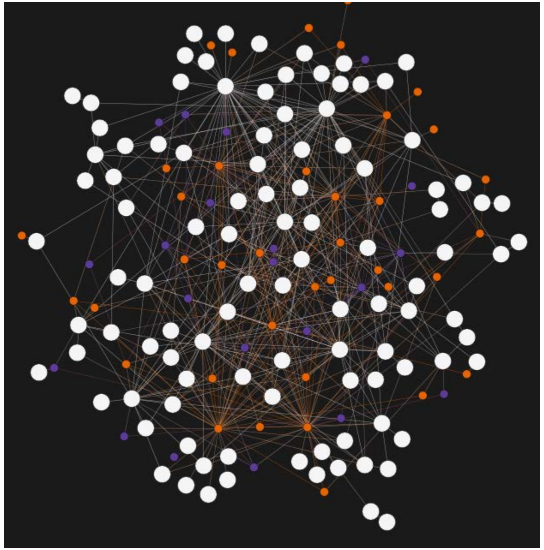
–more likely than

- Benign site → Malicious site



Transform into a Graphical Model

- Model web graph as a Markov random field
  - Vertices: probability of a host being malicious
  - Edges: hyperlinks
- Exploit web link structure
- Use whitelist and blacklist to assign priors for some nodes
- Use uniform prior for rest
- Edge factors are assigned based on domain knowledge
- Perform inference (run BP) to estimate marginal probability
- Estimate score and rank



Number/type of nodes not representative of real graph

Challenges of running BP on Large-Scale Graphical Models

- Large Size/Random access
  - Use large memory machine to keep graph in memory
- Communication bound
  - Use shared memory
- Parallelization
  - Synchronous processing (BSP)
- Partitioning
  - Heuristic to minimize vertex replication

Implementation

Requirements

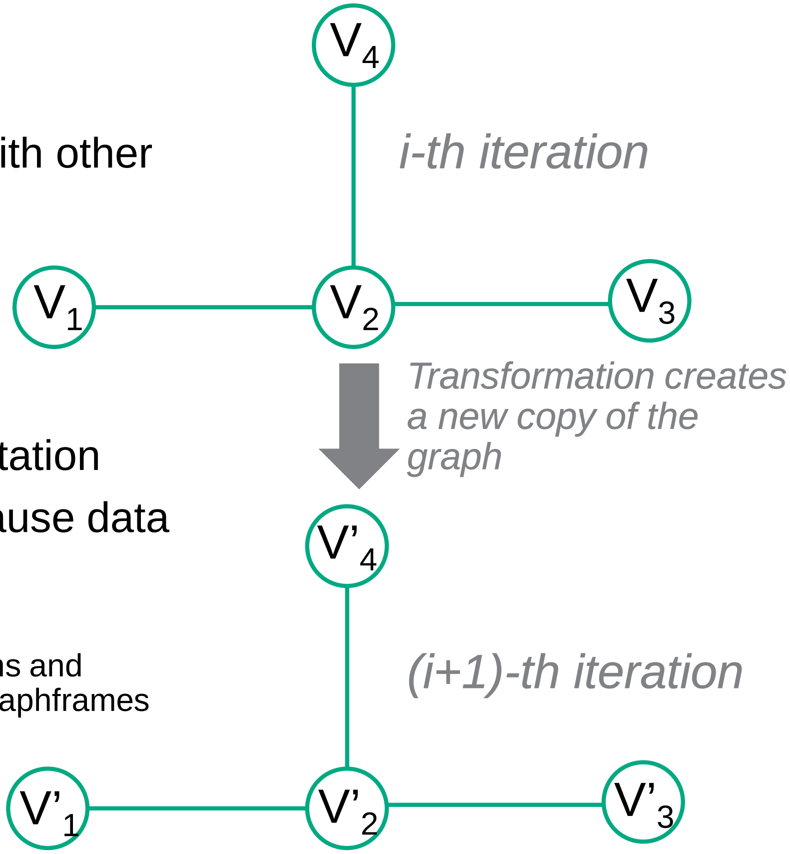
- Scalability
  - Apache Spark GraphX
- Generalized graph representation
  - Factor graph format
  - Handles factor of any order and variable domain of any size
- Numerical stability
  - Log domain calculations to delay underflow (overflow)
  - Factor math

Improvements

- Communication; memory

Scalability with Apache Spark GraphX

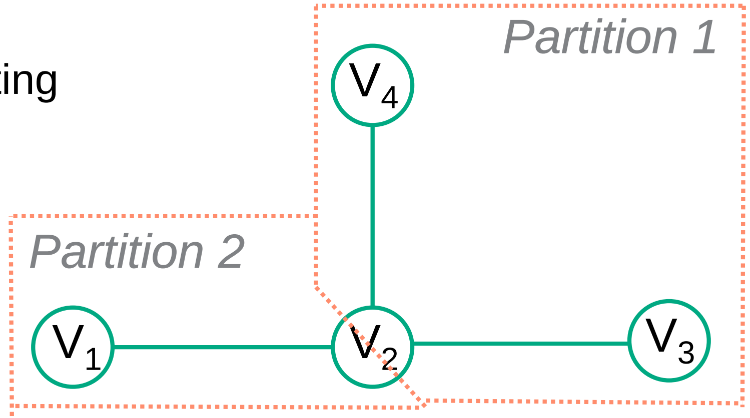
- Pros
  - Provided by Apache Spark => easy to combine with other big data workloads
  - Convenient message scatter/gather API
- Cons
  - Is not actively developed
  - Large memory overhead due to internal representation
  - Requires 2X memory for iterative algorithms because data is immutable in Spark
    - Graph on previous iteration and graph on this iteration
    - Graphframes library does not properly support iterative algorithms and outsources them to GraphX. Will move our implementation to Graphframes once this issue is addressed



Graph representation in GraphX

- Graph in GraphX: VertexRDD, EdgeRDD, routing table
  - Vertex cut - replicated vertices
  - Routing table - data structure for updating replicated vertices
- Spark's computation model (BSP):

$$t = t_{cp} + t_{cm}$$



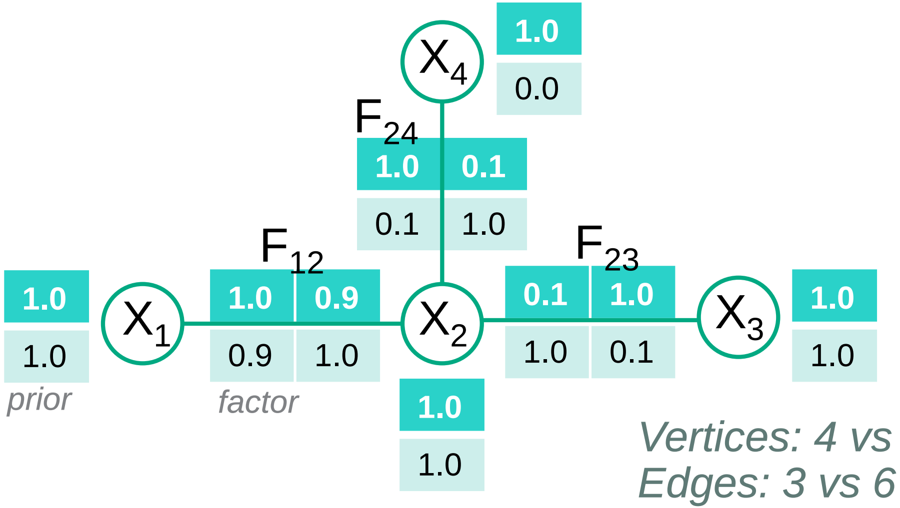
- Estimations for Belief Propagation

$$t_{cp} = max_{i \in [1,n]}(E_i)/(F \cdot n) \cdot (S + 2 \cdot (S + S^2)) \quad t_{cm} = 32/B \cdot r \cdot V \cdot S$$

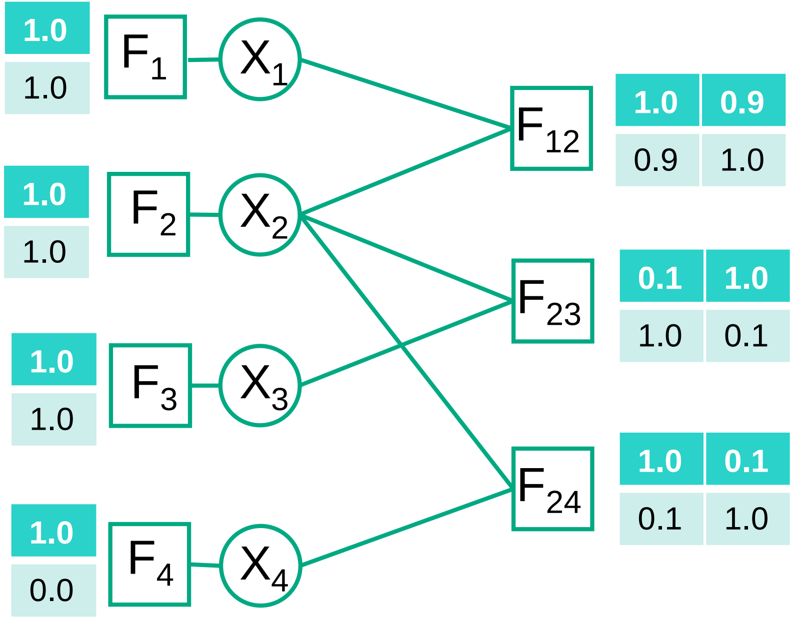
  - *E* - edges, *V* – vertices, *S* – number of states, *F* – FLOPS, *B* – bandwidth, *r* – replication factor
- Insight: communication is proportional to the number of vertices, computation is relatively inexpensive
- More details about the model: "Modeling Scalability of Distributed Machine Learning" *ICDE 2017*

Graph Representation for BP

- Pairwise
  - Variables are represented as vertices
  - Factors are stored on edges
  - Number of vertices == number of variables
  - Number of edges == number of factors



- Factor graph (convenient in many domains)
  - Can represent higher-order factors (can be converted to pairwise)
  - Variables and factors are represented as vertices
  - Priors are factors, but can be merged to variable vertex
  - Number of vertices == number of variables + number of factors
  - Number of edges == 2 [if factors pairwise] * the number of factors

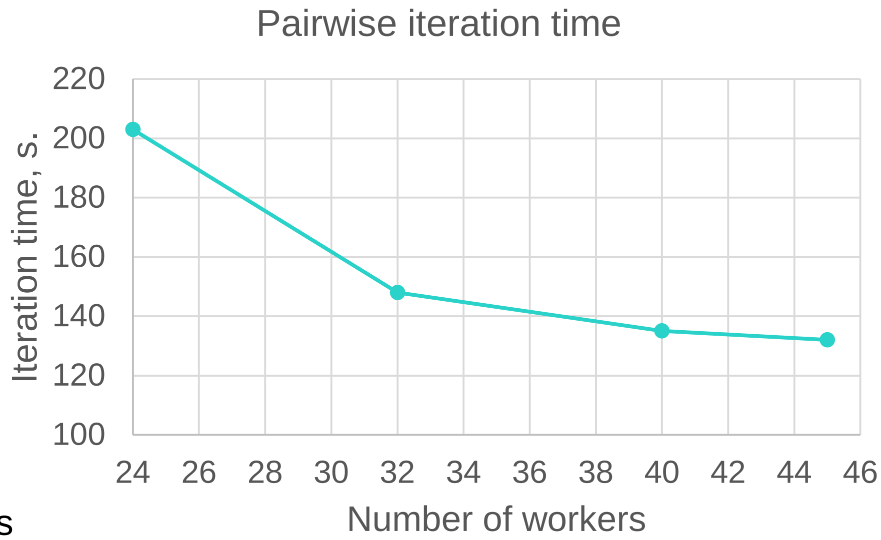


Vertices: 4 vs 7
Edges: 3 vs 6

Numeric example from Chapter 7 of 2013 MIT EECS course 6.869, Bill Freeman and Antonio Torralba

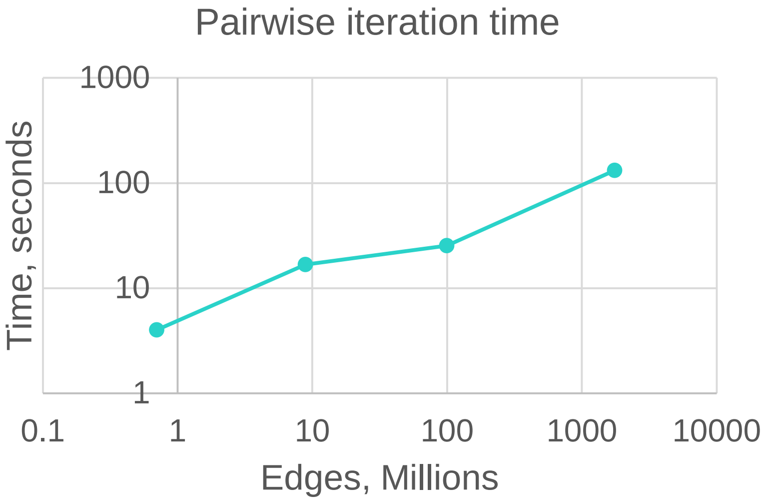
Experiments: malicious web sites detection

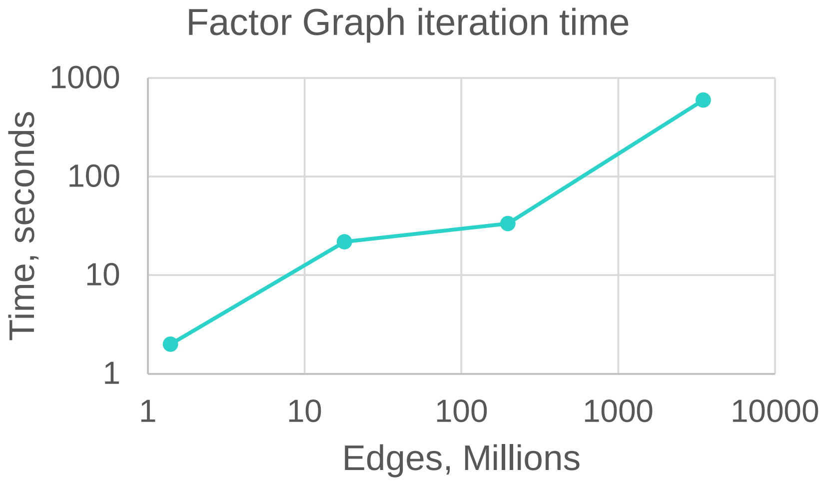
- Dataset
  - Host graph from commoncrawl/web-data-commons
  - 101.7M hosts, 1.75B edges
- Conversion for use-case
  - Host has two states: malicious and normal
  - Host priors from white and black lists
  - Whitelist Alexa500 (17.8M hosts in intersection)
  - Blacklist urlblacklist.com (0.93M hosts in intersection)
  - Factor represents link direction
- Goal: estimate probability of a host to be malicious (normal)
- Converged in 20 iterations, 250 seconds
- Result validation
  - Qualitatively on a subset
  - Malicious sites change quickly



- Hardware: SuperDome X
  - 16x Xeon E7-2890 v2 @ 2.80GHz
  - 11TB shared RAM
- BP algorithm on Spark 1.6.1
  - Shuffle dir in tempfs
  - 24 to 45 workers
  - 32 to 164GB RAM per worker
  - 1 core per worker

Experiments: various graphs





Vertices	Edges	Size on disk	Size in Spark	Iteration time, s.	Iterations
0.2M	**0.7M**	0.03GB	0.4GB	**4**	8
1.6M	**8.9M**	0.3GB	5.2GB	**16.8**	8
16.2M	**99.3M**	5.4GB	59.7GB	**25.3**	8
101.7M	**1.75B**	64GB	840.5GB	**132**	20

Vertices	Edges	Size on disk	Size in Spark	Iteration time, s	Iterations
0.9M	**1.4M**	0.05GB	1.2GB	**1.3**	14
10.7M	**18M**	0.6GB	13GB	**21.8**	15
115.5M	**198.5M**	6.9GB	154.2GB	**33.5**	16
1.9B	**3.5B**	118G	2485G	**600**	40

Improvements

- Hewlett Packard Labs project: Spark for large memory machines (Spark4TM)
- Spark communication layer rewritten
  - Shared memory shuffle engine
- Off-heap memory store introduced
  - Mutable data structure
  - Breaks fault-tolerance model, however
    - The computation takes 10s of minutes at maximum; We can do checkpointing
- Clever graph partitioning
  - Based on heuristic to minimize vertex replication factor
- Spark4TM (custom shuffling + off heap + partitioning) provides 9 s. per iteration on 101M graph vs 132 s. in vanilla Spark
  - Code: https://github.com/HewlettPackard/sparkle

Project “Sandpiper”

– Open source implementation
  – https://github.com/HewlettPackard/sandpiper
  – https://spark-packages.org/package/HewlettPackard/sandpiper

– Example of use

– BP for pairwise factors
  – Two separate for files variables and factors
  – Each record on a separate line
  – Loading is automatically parallelized

File format: variables (id prior)

```
1 1.0 1.0
2 1.0 0.0
...
```

File format: factors (id1 id2 factor table in column major format)

```
1 2 1.0 0.9 0.9 1.0
2 3 0.1 1.0 1.0 0.1
...
```

Scala

```
import sparkle.graph._
val graph = PairwiseBP.loadPairwiseGraph(sc, variableFile, factorFile)
val beliefs = PairwiseBP(graph, maxIterations = 50, epsilon = 1e-3)
```

Example of use: Factor Graph BP

– BP for factor graph
  – libDAI file format with explicit factor ID
  – Factor ID enables splitting the file into parts for loading parallelism

File format

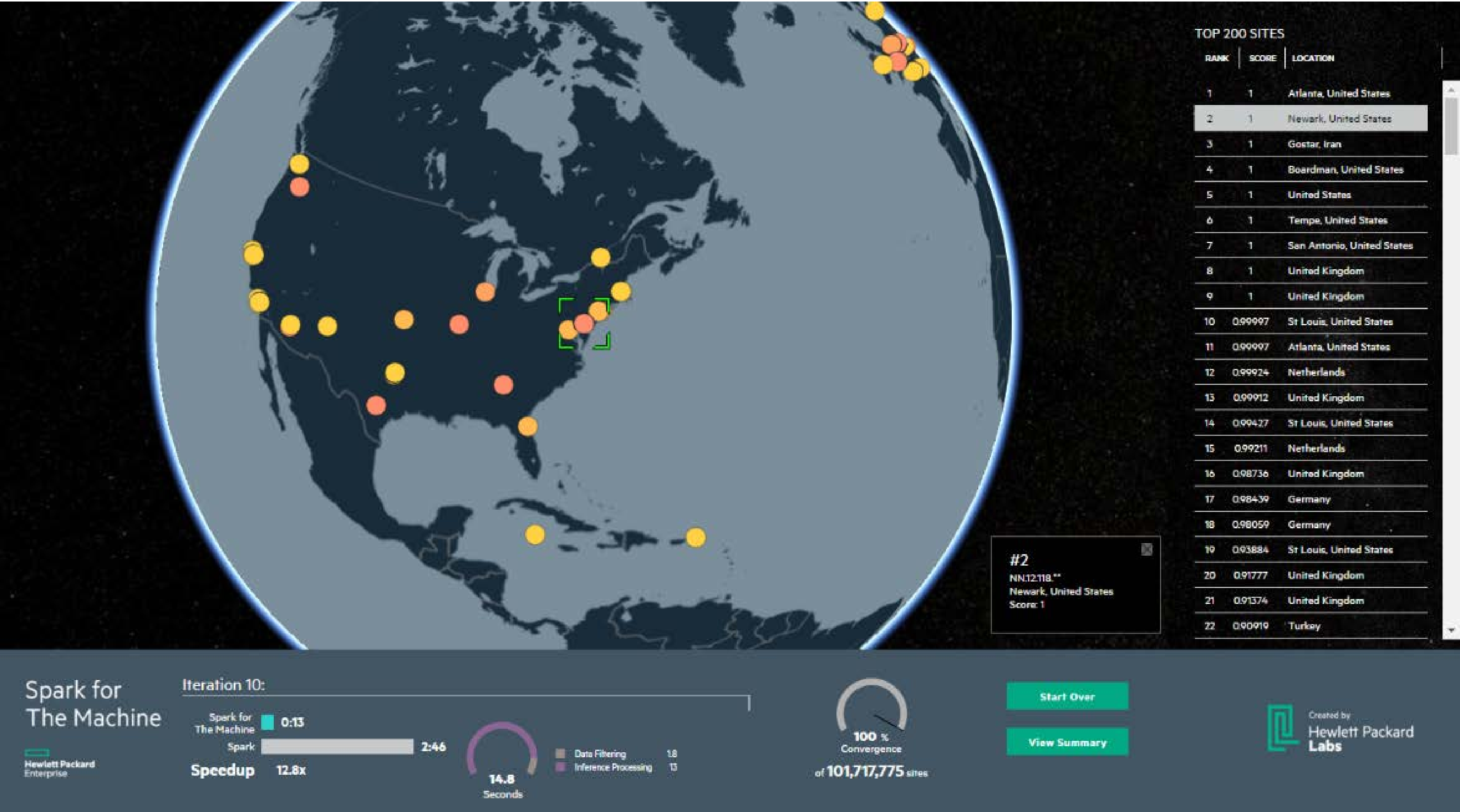
```
# number of factors in the file
7

# factor id preceded with 3 hashes (unique, must not
intersect with variable name/id)
### 5
# number of vars
1
# name of vars
1
# number of values of vars
2
# number of non-zero entries in factor table
2
# non-zero factor table entries
0  1
1  1
```

Scala

```scala
import sparkle.graph._
val graph = Utils.loadLibDAIToFactorGraph(sc, inputPath)
val beliefs = BP(graph, maxIterations = 50, epsilon = 1e-3)
```

Conclusions and Future Work

- Belief propagation algorithm for large graphs
- Application of malicious site detection using BP
- Open source: project “sandpiper”
- Demo at HP Discover
- Future work
  - Re-compute only non-converged nodes
  - Incremental computations
  - Incremental graph construction


TOP 200 SITES
RANK | SCORE | LOCATION
1 1 Atlanta, United States
2 1 Newark, United States
3 1 Gostar, Iran
4 1 Boardman, United States
5 1 United States
6 1 Tempe, United States
7 1 San Antonio, United States
8 1 United Kingdom
9 1 United Kingdom
10 0.99997 St Louis, United States
11 0.99997 Atlanta, United States
12 0.99924 Netherlands
13 0.99912 United Kingdom
14 0.99427 St Louis, United States
15 0.99211 Netherlands
16 0.98736 United Kingdom
17 0.98439 Germany
18 0.98059 Germany
19 0.93884 St Louis, United States
20 0.91777 United Kingdom
21 0.91374 United Kingdom
22 0.90919 Turkey
#2
NN.12.118.**
Newark, United States
Score: 1
Spark for The Machine
Hewlett Packard Enterprise
Iteration 10:
Spark for The Machine 0:13
Spark 2:46
Speedup 12.8x
14.8 Seconds
Data Filtering 1.8
Inference Processing 13
100 % Convergence
of 101,717,775 sites
Start Over
View Summary
Created by Hewlett Packard Labs

Thank you

Spark for large pools of memory

– Make HPE large-memory servers accessible to customers and developers

– Can in-memory analytics perform better with big shared memory?

– Apache Spark as a platform

Superdome X
8 blades x3TB