KnightKing: A Fast Distributed Graph Random Walk Engine

Me: feel bad!

Ke Yang[1], Mingxing Zhang[1,2], **Kang Chen**[1], Xiaosong Ma[3], Yang Bai[4], Yong Jiang[1]

[1] Tsinghua University, [2] Sangfor, [3] QCRI, [4] 4Paradigm

No visa …

Graph Random Walk

Input

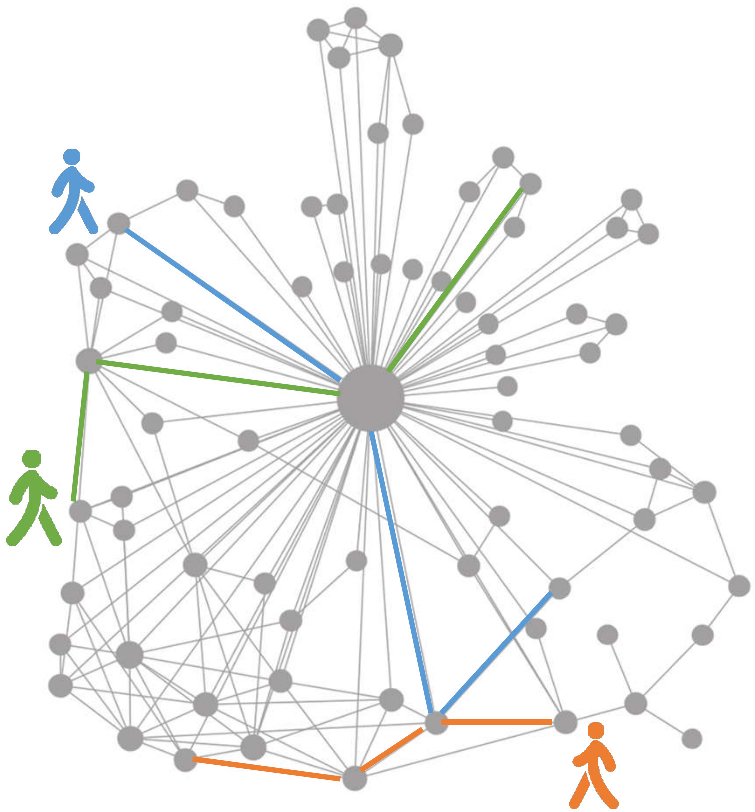
- Graph
- Set of walkers
    - Placed at their starting vertices

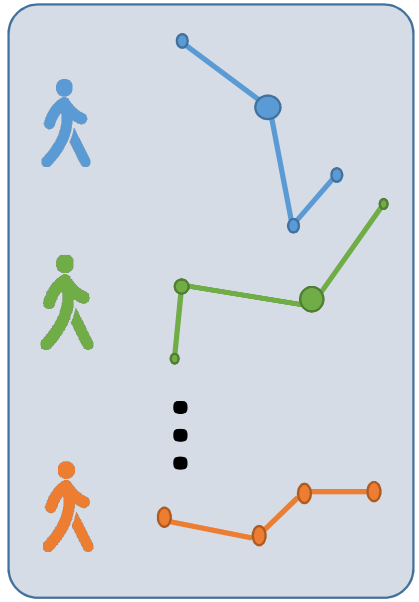
Each walker walks around

- By randomly selecting an edge to follow
- For given number of steps or till given termination condition

Output

- Computation during walk, and/or
- Dump set of walk paths





Increasing Significance of Graph Random Walk

Intuitive way of **extracting information** from graphs

Applications

- ☐ Graph embedding
  - ➢ DeepWalk
  - ➢ node2vec
- ☐ Graph neural network
  - ➢ PinGraph
  - ➢ NetGAN
- ☐ Graph processing
  - ➢ Graph sampling
  - ➢ Vertex ranking

…

Academia

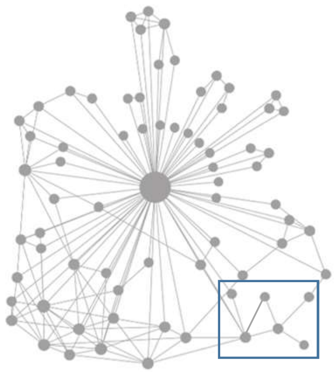
~1700 papers published in 2018 on random walk
(source: Microsoft Academia)

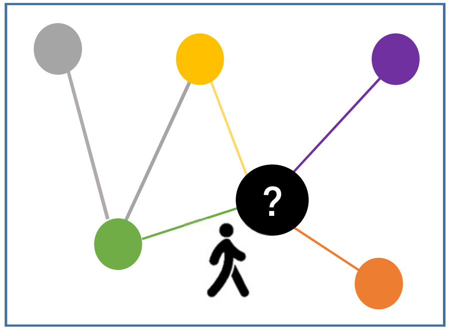
Industry

Used by major companies

…

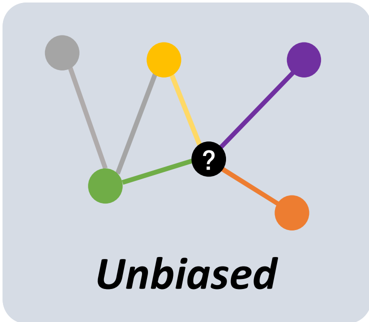
Different Types of Random Walk Algorithms



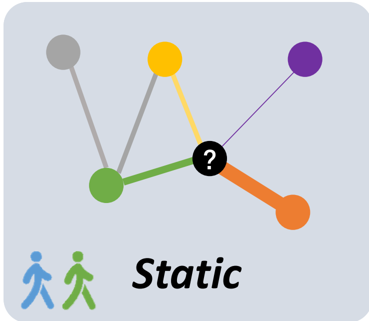


Common to all walking algorithms: Sampling one edge according to ***edge transition probability***
(usually given in un-normalized manner)

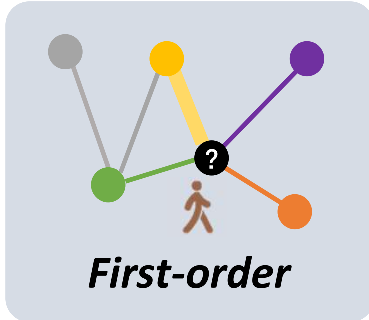
Categories of random walk algorithms



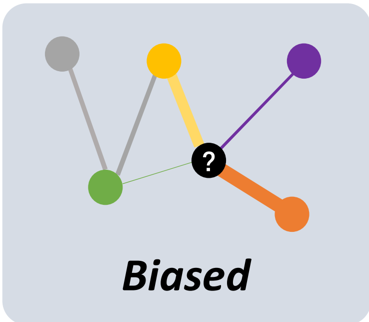
Probability uniform across edges



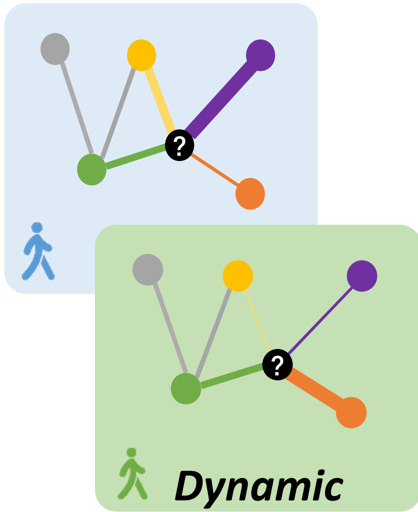
Probability fixed during walk



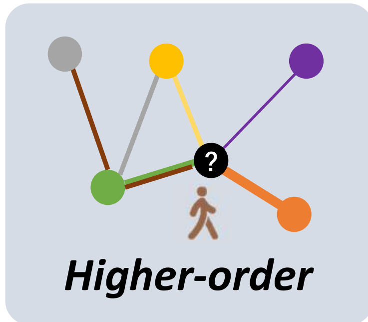
Walk history-oblivious



Probability varied across edges

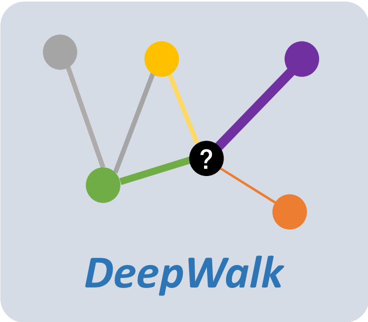


Probability changes during walk and/or depends on walkers



Decision affected by recent steps

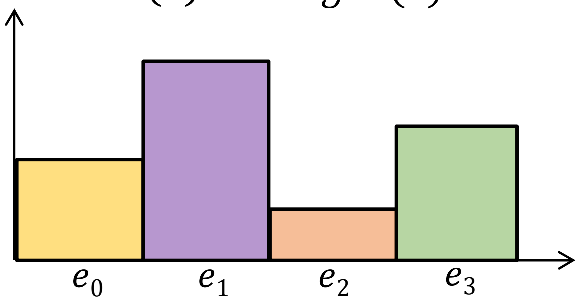
Sample Graph Random Walk Algorithms



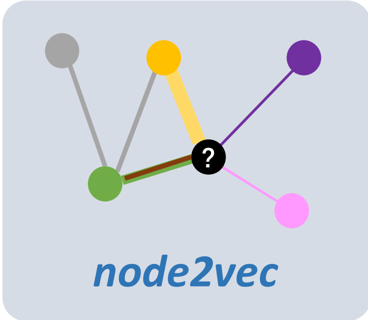
Biased, static, first-order

Edge transition probability:

$$P(e) = weight(e)$$



The probability bars at this black vertex correspond to its edges' thickness



Biased, dynamic, second-order

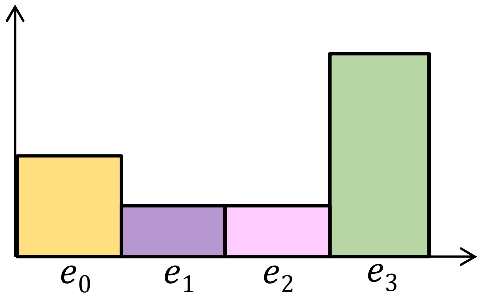
Edge transition probability:

$$P(e) = weight(e) \cdot \alpha_{pq}$$

$$\alpha_{pq}(t,x) = \begin{cases} 1/p, & if\ d_{tx} = 0 \\ 1, & if\ d_{tx} = 1 \\ 1/q, & if\ d_{tx} = 2 \end{cases}$$

Three cases for α: depends on other end of edge: (1) ● (2) ● (3) ● ●

p and q constant hyper-parameters



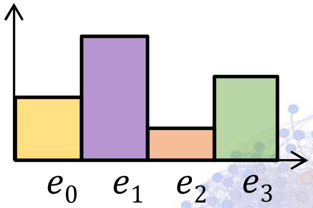
Transition probability

$(p = 0.5,\ q = 2)$

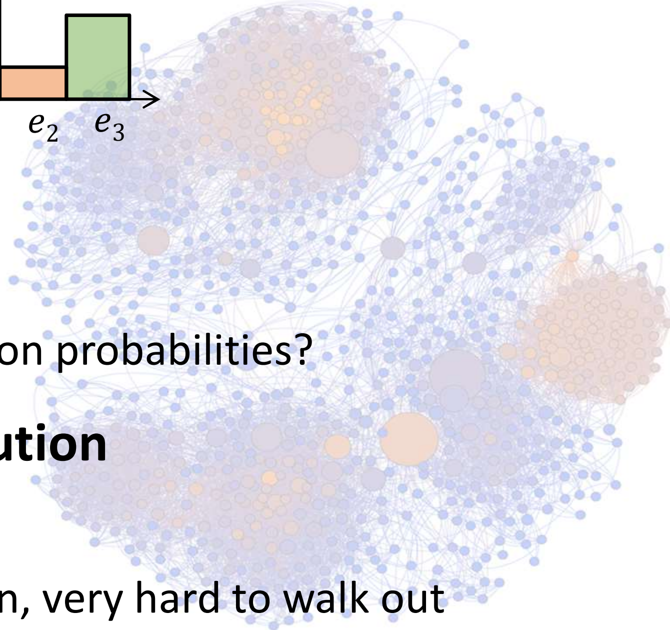
Favoring return edge over new neighborhood

Edge Sampling Can Be Expensive

- Edge sampling is essentially bulk of work



- Dynamic walk: spend lot of time on edge scans
  - To re-compute edge probability distributions
  - Save time by pre-computing and caching all possible transition probabilities?
- Real-world graphs have **highly skewed degree distribution**
  - Small subset of vertices attract majority of edges
  - These hot spots become “walker traps”: super easy to step in, very hard to walk out



Graph	Vertices	Edges (undirected)	Graph size	Index storage	Degree mean	Degree variance	Avg. # of edges checked per step
Twitter	41.7M	2.93B	22GB	980TB	70.4	6.4E6	92202
UK-Union	134M	9.39B	70GB	1481TB	70.3	3.0E6	47790

Pre-compute for node2vec

Our Work: Fast Graph Random Walk Engine

- ***KnightKing***: effortlessly coordinates millions of walkers on large graphs
- First general-purpose engine for graph random walk
  - To enable algorithm expression: Unified edge transition probability definition
  - To speedup walks: Rejection-based, fast and exact edge sampling
  - For programmers: Walker-centric programming model
  - Common optimizations for different random walk algorithms
- Distributed
  - Scale out if needed
- Available at

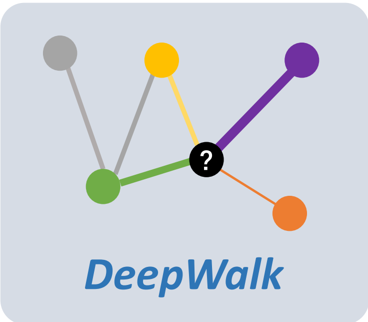
  github.com/KnightKingWalk

Unified Transition Probability Definition

- Key idea: decompose the probability definition to separate static and dynamic components
  - Static: reflecting input graph properties, stays constant
  - Dynamic: reflecting walker preferences or states

$$P = \underbrace{P_s}_{\text{Static component}} \cdot \underbrace{P_d}_{\text{Dynamic component}} \cdot \underbrace{P_e}_{\text{Extension component}}$$

- Examples

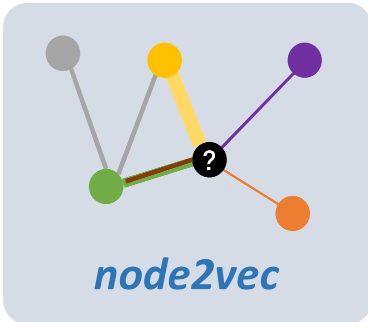


Edge transition probability:

$$P(e) = weight(e)$$

$$P = weight(e) \cdot \cancel{P_d} \cdot P_e$$

(Static walk: trivial dynamic component)



Edge transition probability:

$$P(e) = \alpha_{pq} \cdot weight(e)$$

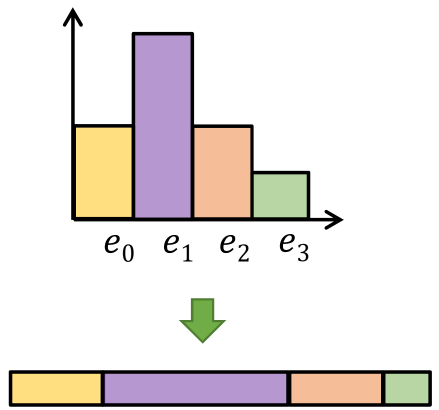
α_{pq}: depends on both graph topology and walk history

$$\alpha_{pq}(t,x) = \begin{cases} 1/p, & if\ d_{tx} = 0 \\ 1, & if\ d_{tx} = 1 \\ 1/q, & if\ d_{tx} = 2 \end{cases}$$

$$P = weight(e) \cdot \alpha_{pq} \cdot P_e$$

Static Walk: Edge Scan Once and For All

- Do edge scan only once, at beginning of run (pre-processing), followed by quick sampling
- KnightKing adopts existing approaches
  - Inverse Transform Sampling (ITS)
    - Uniform sampling in 1-D space, corresponding to per-edge probabilities
    - $O(n)$ time and space to build index array
    - $O(\log(n))$ time to sample edge using binary search
  - Alias Method (see paper for details)
    - A more sophisticated alias table: Splitting per-page probabilities into pieces and construct equal-sum buckets
    - Uniform sampling of buckets, weighted sampling of edges within
    - $O(n)$ time and space to build alias table
    - $O(1)$ time to sample edge

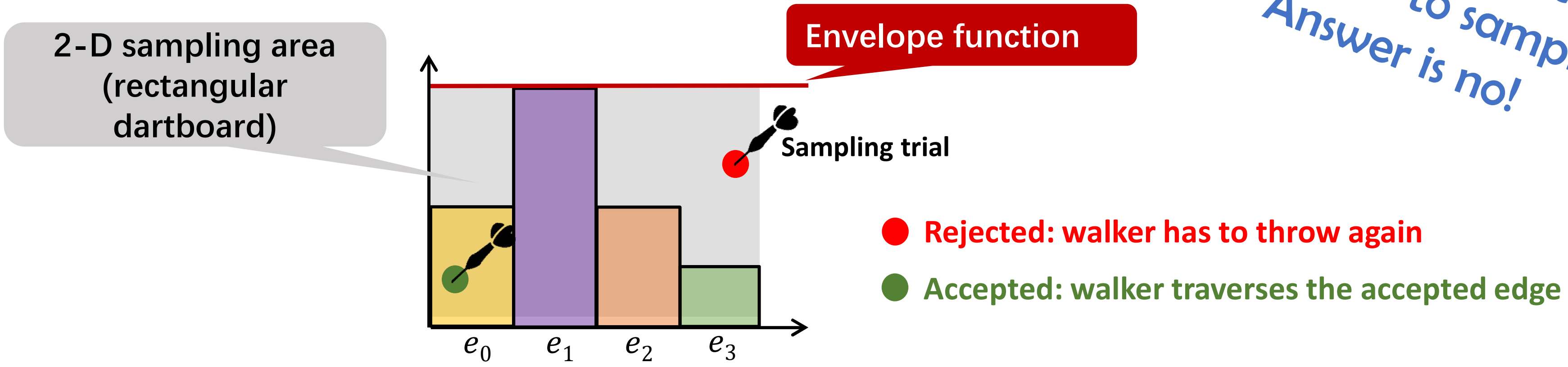


1-D space for ITS sampling

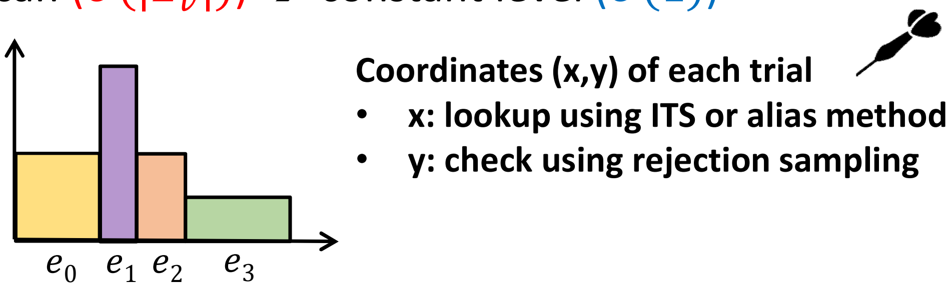
Eliminating Edge Scans During Dynamic Walk

- Key idea: **rejection sampling**
  - Old way: survey ***all edges***, pluck one with appropriate probability
  - Now: sample first, then check ***that and only that edge***

Do we have to go through all edges to sample one?
Answer is no!

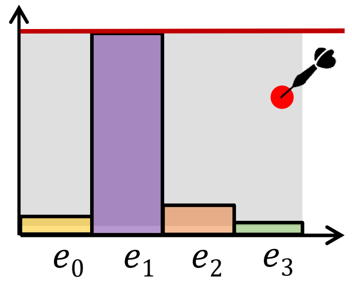


- **Correctness**: the probability of the edges being sampled is equivalent to the relative height of their bars.
- **Efficiency**: reduce sampling overhead, linear scan $(O(|E_v|))$ ➔ constant level $(O(1))$
- Incorporating static component:
  - P_s determines ***widths*** of bars
  - P_d determines ***heights*** of bars

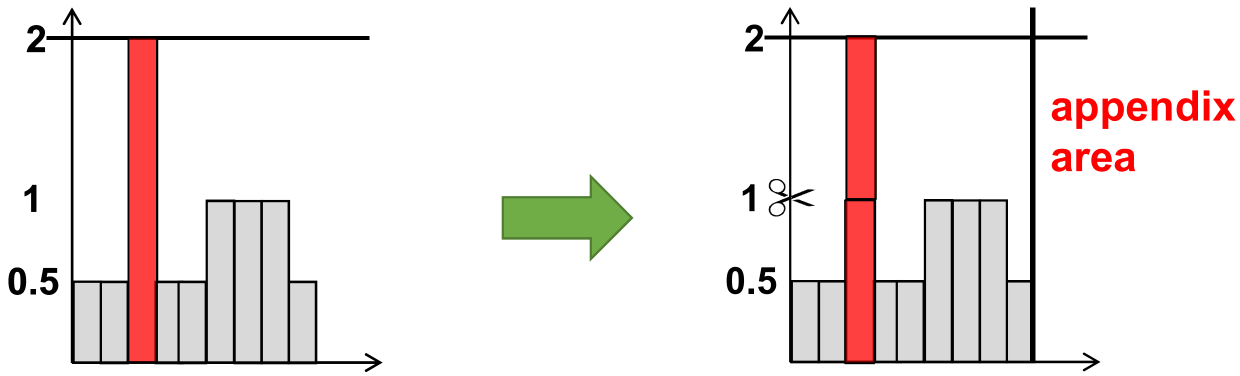


Coordinates (x,y) of each trial
- **x: lookup using ITS or alias method**
- **y: check using rejection sampling**

Optimization: More Efficient Dartboard (I)



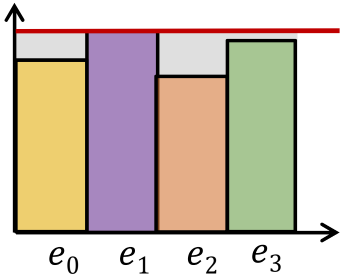
- Performance depends on **efficiency of dartboard**
  - Tighter envelop, smaller white area, fewer trials
  - Bad case: very few tall outliers push up entire envelope
    - Worse for high-degree vertices
    - E.g., node2vec, assigns high probability to single “return edge”
- KnightKing optimization: ***folding***
  - Optional APIs to identify <u>transition probability outliers</u>



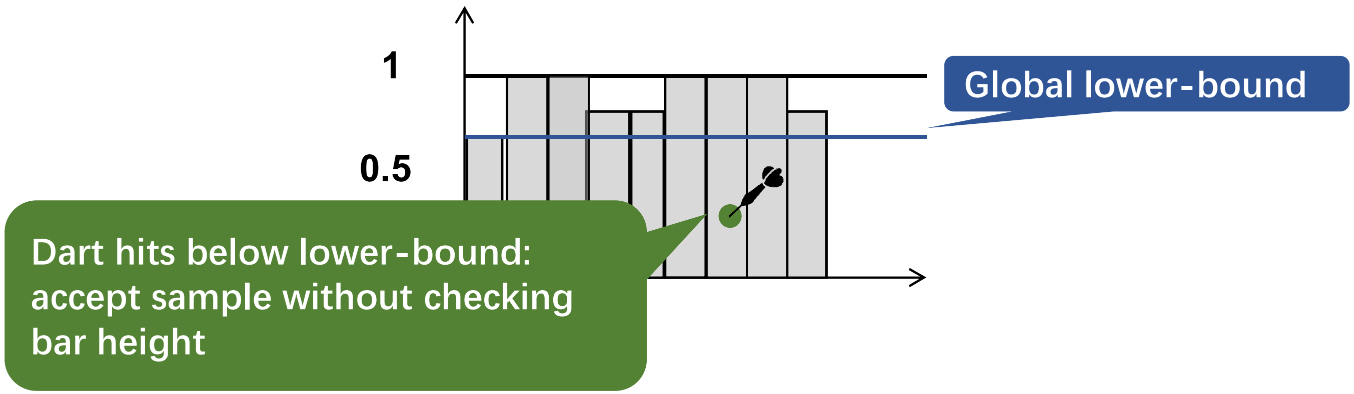
- Cut outliers, put cropped segments to right side of board as **appendix area**
- Lower down envelope

Optimization: More Efficient Dartboard (II)

- Super tight envelope good? Wasteful too!
  - NightKing never builds physical dartboard
  - After each trial, edge sampled, dynamic compute bar height
    - Could involve inter-node communication, expensive!



- KnightKing optimization: ***lower-bound based early acceptance***
  - Optional APIs to mark global lower-bound
  - Most darts hit below lower-bound line



Walker-centric Programming Model and APIs

Gemini GraphLab GIRAPH Spark GraphX

KnightKing

Graph engines: vertex-centric

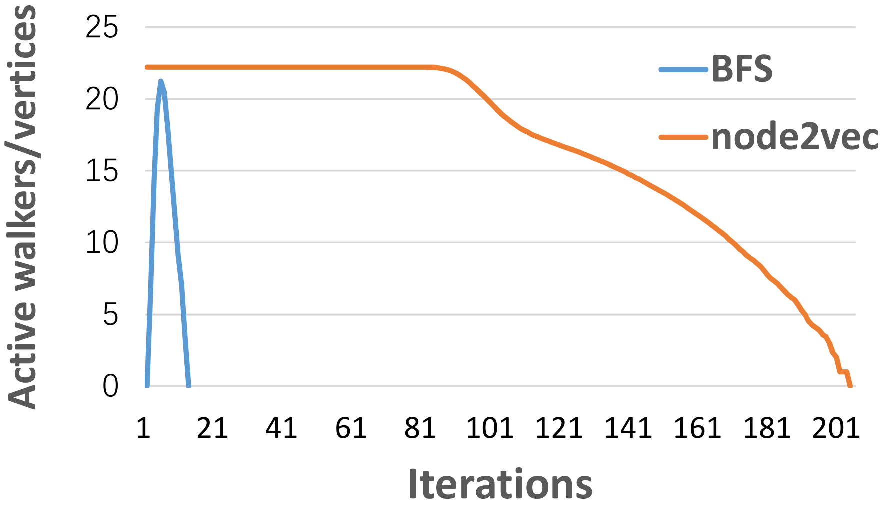
- Vertex states
  - Initial
  - How to update
- Actions (update propagation)
  - Message content generation
  - State update upon receiving message
  - User-optional optimization
    - Enable push/pull hybrid mode (optional)
  - Transparent optimizations by framework
- Termination condition

Random walk engine: walker-centric

- Walker states
  - # of walkers
  - Start positions and initial states
- Actions (walk)
  - Edge transition probability
    - Static and dynamic
    - Envelope for rejection sampling
  - Queries for higher-order walks
  - User-optional optimization
    - Outlier, lower-bound specification
  - Transparent optimizations by framework
- Termination condition

System Design and Implementation

- C++, core code about 2500 lines
- Design choices
  - BSP computation model, 1-D graph partitioning, CSR for in-memory graph storage, OpenMPI for message passing
- Pipeline and scheduling optimizations specifically targeting distributed graph random walk (see paper for details)
  - Straggler problem
  - Different walk speed
  - More severe imbalance



Evaluation Setup

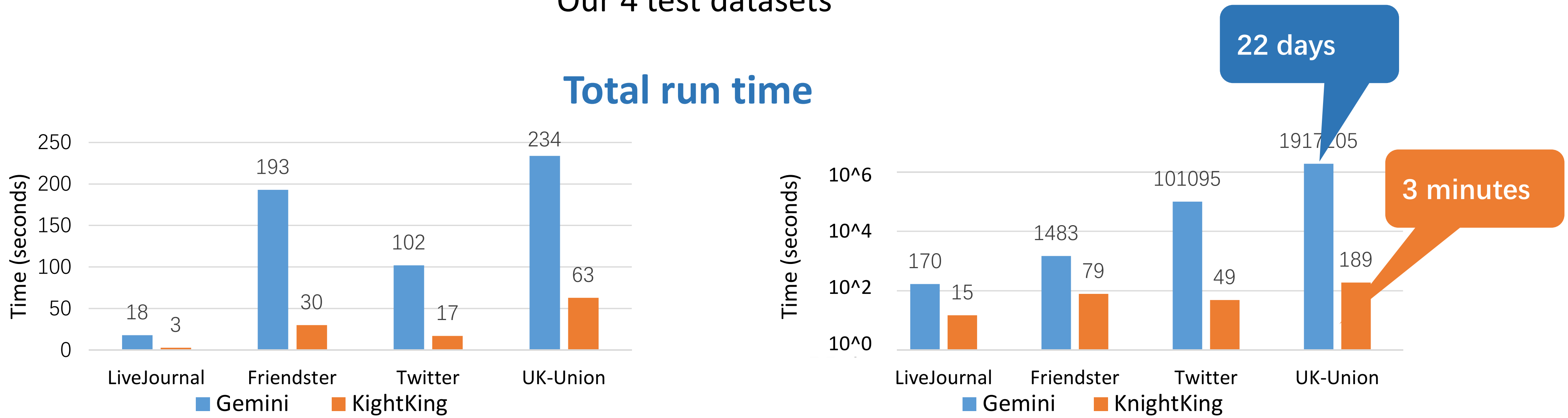
- Environment
  - 8-node cluster with 40Gbps InfiniBand interconnection
  - Each node has 2 8-core 2GHz Intel Xeon, 20MB L3 cache, and 94GB DRAM
- Dataset
  - 4 real world graphs
  - Synthetic graphs with different metrics
- Applications
  - DeepWalk, Personalized PageRank, meta-path random walk, node2vec
- Baseline
  - Implement prior sample methods with full-edge-scan on Gemini [OSDI16]
    - significantly out-performs existing available single-algorithm random walk implementations

Benchmark and Overall Performance

Graph	LiveJournal	Friendster	Twitter	UK-Union
Vertices	4.85M	70.2M	41.7M	134M
Edges	69.0M	1.81B	1.47B	5.51B
Degree Variance	2.72E3	1.62E4	6.42E6	3.04E6

Our 4 test datasets

Total run time



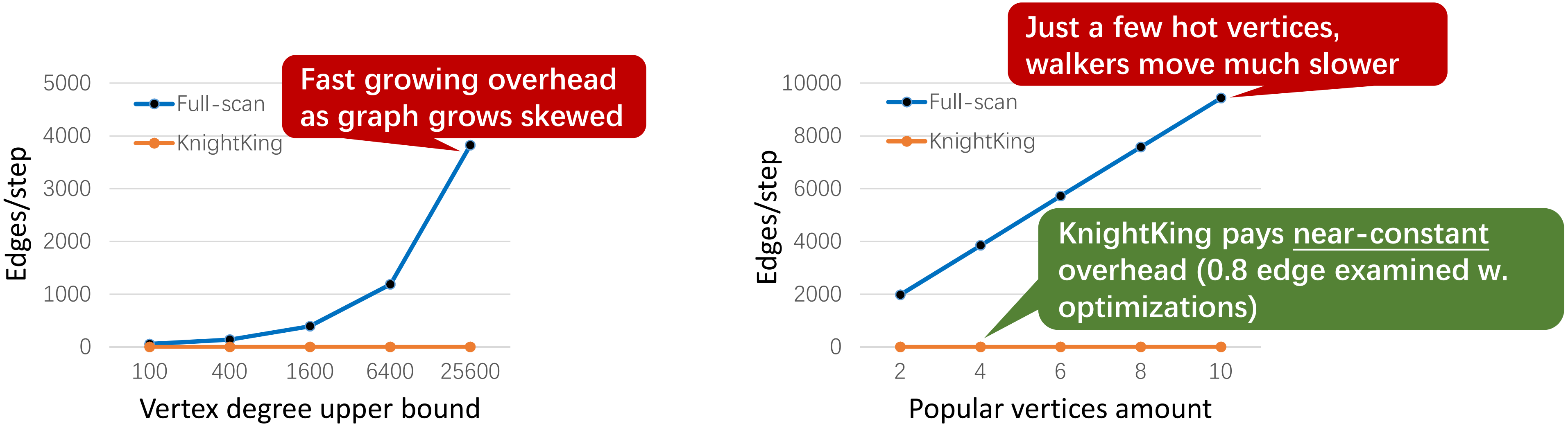
DeepWalk on weighted graph
($|V|$ walkers, 80 steps each)

node2vec on weighted graph (base-10 log scale)
($|V|$ walkers, 80 steps each)

Graph Topology Sensitivity

KnightKing ***insensitive to graph topology***, unlike existing method

Graph	Vertices	Degree mean	Degree variance
Truncated power-law	10 M	51~159	3.4E2~7.1E5
Several popular vertices	10 M	100~101	2.0E5~1.0E6



(a) Truncated power-law distribution

(b) Small number of million-edge vertices

Node2vec sampling overhead on synthetic graphs: <u>average number of edges examined , per walker per step</u>

Conclusion

- Dynamic, higher-order walks not as expensive as people previously believed
  - Exact, constant-time sampling possible with rejection sampling

- People could use general-purpose random walk engine
  - Just like we use graph engines
  - Easy algorithm implementation, common optimizations
  - Hidden communication/scheduling details

Thank you!

Check out at *github.com/KnightKingWalk*