STAR: Steiner Tree Approximation in Relationship Graphs

Gjergji Kasneci

Joint Work with:
Maya Ramanath, Mauro Sozio,
Fabian M. Suchanek, and Gerhard Weikum
Relationship Graphs

- Simple, flexible, explicit way to represent knowledge
- Semantics encoded by node and edge labels
- Edge weights may represent connectivity strengths
- Examples:
  - Roadmaps
  - Social networks
  - Biochemical networks
  - General purpose ontologies (e.g. WordNet, SUMO, Cyc, YAGO, ...)
  - ...
Slightly complex biochemical network
Informal Problem Definition

• General Task:
  Knowledge discovery as opposed to mere look-up

• Scenario:
  Find efficiently the closest connection between any given entities
Informal Problem Definition

• General Task:
  Knowledge discovery as opposed to mere look-up

• Scenario:
  Find efficiently the closest connection between any given entities

• Examples:
  **Encyclopedic queries**
  What do Jackie Chan, Jules Verne, and Shirley MacLaine have in common?

  **Criminalistic queries**
  What do John Gotti, Paul Castellano, and Carlo Gambino have in common?

  **Biomedical queries**
  What is the relation between Glutamines and Amino Acids?
Problem Definition

• Given:
  – Relationship graph $G$
  – $l \geq 2$ entities (query entities or query nodes),
  – a cost function $w(g) = \sum_{e \in E(g)} d(e)$, for every subgraph $g \subseteq G$

• Task:
  – Find a min-cost subtree of $G$ that interconnects all query entities
  
  - Steiner Tree Problem (NP-hard)
  - Tons of literature and solutions
  
  – Find top-k min-cost subtrees that interconnect all query nodes
Distance Network Heuristic
1) Build complete graph on query nodes (an edge represents shortest path between its end nodes)
2) Use MST heuristic to find a solution

Approaches:
DNH [Kou et al.; AI 1981]
FDNH [Mehlhorn et al.; IPL 1988]
BANKS I [Bhalotia et al.; ICDE’02]
BANKS II [Kacholia et al.; VLDB’05]
Related Work

**Distance Network Heuristic**
1) Build complete graph on query nodes (an edge represents shortest path between its end nodes)
2) Use MST heuristic to find a solution

**Approaches:**
- **DNH** [Kou et al.; AI 1981]
- **FDNH** [Mehlhorn et al.; IPL 1988]
- **BANKS I** [Bhalotia et al.; ICDE’02]
- **BANKS II** [Kacholia et al.; VLDB’05]

**Dynamic Programming**
1) Compute optimal results for all subsets of the query nodes
2) Infer optimal result for all query nodes

**Approaches:**
- **D&W** [Dreyfus & Wagner; NJ 1981]
- **DPBF** [Ding et al.; ICDE’07]
Related Work

Distance Network Heuristic
1) Build complete graph on query nodes (an edge represents shortest path between its end nodes)
2) Use MST heuristic to find a solution

Approaches:
DNH [Kou et al.; AI 1981]
FDNH [Mehlhorn et al.; IPL 1988]
BANKS I [Bhalotia et al.; ICDE’02]
BANKS II [Kacholia et al.; VLDB’05]

Span and Cleanup
1) Start to build an MST from a query node, until all query nodes are covered
2) Delete redundant nodes

Approaches:
RIU [W.-S. Li et al.; TKDE’02]
IHLER [Ihler; WG 1991]
R&W [Reich & Widmeyer; WG 1989]

Dynamic Programming
1) Compute optimal results for all subsets of the query nodes
2) Infer optimal result for all query nodes

Approaches:
D&W [Dreyfus & Wagner; NJ 1981]
DPBF [Ding et al.; ICDE’07]
Related Work

Distance Network Heuristic
1) Build complete graph on query nodes (an edge represents shortest path between its end nodes)
2) Use MST heuristic to find a solution

Approaches:
**DNH** [Kou et al.; AI 1981]
**FDNH** [Mehlhorn et al.; IPL 1988]
**BANKS I** [Bhalotia et al.; ICDE’02]
**BANKS II** [Kacholia et al.; VLDB’05]

Dynamic Programming
1) Compute optimal results for all subsets of the query nodes
2) Infer optimal result for all query nodes

Approaches:
**D&W** [Dreyfus & Wagner; NJ 1981]
**DPBF** [Ding et al.; ICDE’07]

Span and Cleanup
1) Start to build an MST from a query node, until all query nodes are covered
2) Delete redundant nodes

Approaches:
**RIU** [W.-S. Li et al.; TKDE’02]
**IHLER** [Ihler; WG 1991]
**R&W** [Reich & Widmeyer; WG 1989]

Partition and Index
1) Partition graph into blocks
2) Build inter-block and intra-block shortest path indexes

Approaches:
**BLINKS** [H. He et al.; SIGMOD’07]
**EASE** [G. Li et al.; SIGMOD’08]
Related Work

Distance Network Heuristic
1) Build complete graph on query nodes (an edge represents shortest path between its end nodes)
2) Use MST heuristic to find a solution

Approaches:
DNH [Kou et al.; AI 1981]
FDNH [Mehlhorn et al.; IPL 1988]
BANKS I [Bhalotia et al.; ICDE’02]
BANKS II [Kacholia et al.; VLDB’05]

Dynamic Programming
1) Compute optimal results for all subsets of the query nodes
2) Infer optimal result for all query nodes

Approaches:
D&W [Dreyfus & Wagner; NJ 1981]
DPBF [Ding et al.; ICDE’07]

Span and Cleanup
1) Start to build an MST from a query node, until all query nodes are covered
2) Delete redundant nodes

Approaches:
RIU [W.-S. Li et al.; TKDE’02]
IHLER [Ihler; WG 1991]
R&W [Reich & Widmeyer; WG 1989]

Partition and Index
1) Partition graph into blocks
2) Build inter-block and intra-block shortest path indexes

Approaches:
BLINKS [H. He et al.; SIGMOD’07]
EASE [G. Li et al.; SIGMOD’08]

STAR: Combination of Heuristics + Local Search
## Related Work

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Performance Ratio</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLINKS [H. He et al.; SIGMOD’07]</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>R&amp;W [Reich &amp; Widmayer; WG 1989]</td>
<td>unbounded</td>
<td>(O(l \cdot (m + n \log n)))</td>
</tr>
<tr>
<td>Ihler [WG 1991]</td>
<td>(O(l))</td>
<td>(O(l \cdot n \cdot (m + n \log n)))</td>
</tr>
<tr>
<td>BANKS-I [Bhalotia et al.; ICDE’02]</td>
<td>(O(l))</td>
<td>(O(n^2 \log n + n \cdot m))</td>
</tr>
<tr>
<td>BANKS-II [Kacholia et al.; VLDB’05]</td>
<td>(O(l))</td>
<td>(O(n^2 \log n + n \cdot m))</td>
</tr>
<tr>
<td>RIU [W.-S. Li et al.; TKDE’02]</td>
<td>(O(l))</td>
<td>(O(l \cdot n \cdot (m + n \log n)))</td>
</tr>
<tr>
<td>Bateman et al. [ISPD 1997]</td>
<td>(O((1 + \ln(l/2)) \cdot \sqrt{l}))</td>
<td>(O(n^2 \cdot l^2 \log l))</td>
</tr>
<tr>
<td>Charikar et al. [JA 1999]</td>
<td>(O(i(i-1)i^{l/i}))</td>
<td>(O(n^i \cdot l^{2i}))</td>
</tr>
<tr>
<td>STAR</td>
<td>(O(\log(l)))</td>
<td>(O(\frac{w_{\text{max}}}{\varepsilon \cdot w_{\text{min}}} \cdot m \cdot l \cdot (n \log n + m)))</td>
</tr>
<tr>
<td>DNH [Kou et al.; AI 1981]</td>
<td>(O(2(1−1/l)))</td>
<td>(O(n^2 \cdot l))</td>
</tr>
<tr>
<td>DPBF [Ding et al.; ICDE’07]</td>
<td>optimal</td>
<td>(O(3^l n + 2^l ((l + \log n)n + m)))</td>
</tr>
</tbody>
</table>

\(n\): # nodes in \(G\) \hspace{1cm} \(m\): # edges in \(G\) \hspace{1cm} \(l\): # query terms \hspace{1cm} \(i\): tree depth
Outline

 Intro & Related Work

• STAR:
  – Algorithm
  – Heuristics
  – Analysis
  – Top-\(k\)

• Experiments

• Conclusion
STAR: A Metaheuristic

• 1. Phase:
  – Construct an initial tree as quickly as possible, e.g. by:
    • exploiting meta information about the graph
    • exploiting heuristics for fast search space traversal
    • careful precomputation of interconnecting paths (at least for some nodes)
STAR: A Metaheuristic

• 1. Phase:
  – Construct an initial tree as quickly as possible, e.g. by:
    • exploiting meta information about the graph
    • exploiting heuristics for fast search space traversal
    • careful precomputation of interconnecting paths (at least for some nodes)

• 2. Phase:
  – Improve current solution iteratively and quickly by replacing it with better solutions from its local neighborhood, e.g. by:
    • effectively pruning the local neighborhood
    • exploiting heuristics for fast search space traversal
STAR: Phase I

- Often relationship graphs come with taxonomic backbone (e.g. WordNet, SUMO, Cyc, YAGO, ...)

STAR: Phase I

• Often relationship graphs come with taxonomic backbone (e.g. WordNet, SUMO, Cyc, YAGO, ...)

• Build an initial tree by exploiting this taxonomic info

• Follow only type and subClassOf edges to taxonomic ancestor of query entities
STAR: Phase I

- Often relationship graphs come with taxonomic backbone (e.g. WordNet, SUMO, Cyc, YAGO, ...)
- Build an initial tree by exploiting this taxonomic info
- Follow only type and subClassOf edges to taxonomic ancestor of query entities

→ Very few edges to visit,
→ Very efficient
Example: Phase I
STAR: Phase I

• When no taxonomic info available:
  – Fast search space traversal
    • Use breadth-first iterators starting from each query nodes
    • Return an initial tree as soon as the iterators meet
      → Much faster than using single-source-shortest-path iterators (BANKS strategy)
STAR: Phase II

- Improve current tree as quickly as possible with better solutions from local neighborhood

Algorithm 1: improve(T)

Q: priority queue of replaceable paths in T
//ordered by decreasing weights

while Q.notEmpty() do
    p = Q.dequeue()
    \(\{T_1, T_2\}\) = Remove(p, T)
    findShortestPath(T_1, T_2)
    //shortest path between \(T_1\) and \(T_2\) in G

    if \(w(T') < w(T)\) then
        \(T = T'\)
        Q: priority queue of replaceable paths in T
        //ordered by decreasing weights
    end if
end while
return T

Fast pruning of local neighborhood
STAR: Phase II

- Improve current tree as quickly as possible with better solutions from local neighborhood

**Algorithm 1: improve(T)**

Q: priority queue of replaceable paths in T  
//ordered by decreasing weights

while Q.notEmpty() do
  p = Q.dequeue()
  \{T\_1, T\_2\} = Remove(p, T)
  findShortestPath(T\_1, T\_2)
  //shortest path between T\_1 and T\_2 in G

  if w(T') < w(T) then
    T = T'
    Q: priority queue of replaceable paths in T  
    //ordered by decreasing weights
  end if
end while

return T

Fast pruning of local neighborhood

Which paths are replaceable?
STAR: Phase II

- **Definitions:**
  1. **Fixed node:** *either a query node or a node of degree > 2 in the current tree*
  2. **Loose path:** *path of the current tree in which only end nodes are fixed nodes*

**Algorithm 1:** improve(T)

Q: priority queue of loose paths in T  
//ordered by decreasing weights

while Q.notEmpty() do
  p = Q.dequeue()
  \{T_1, T_2\} = Remove(p, T)
  findShortestPath(T_1, T_2)  
  //shortest path between T_1 and T_2 in G

  if w(T') < w(T) then
    T = T'
    Q: priority queue of loose paths in T  
    //ordered by decreasing weights
  end if
end while
return T
STAR: Phase II

• Definitions:
  (1) Fixed node: either a query node or a node of degree >2 in the current tree
  (2) Loose path: path of the current tree in which only end nodes are fixed nodes

Algorithm 1: improve(T)

Q: priority queue of loose paths in T
  //ordered by decreasing weights

while Q.notEmpty() do
  p = Q.dequeue()
  \{T_1, T_2\} = Remove(p, T)
  findShortestPath(T_1, T_2)
  //shortest path between T_1 and T_2 in G
  if w(T’) < w(T) then
    T = T’
    Q: priority queue of loose paths in T
    //ordered by decreasing weights
  end if
end while
return T
STAR: Phase II

• **Definitions:**
  
  (1) **Fixed node:** either a query node or a node of degree >2 in the current tree
  
  (2) **Loose path:** path of the current tree in which only end nodes are fixed nodes

**Algorithm 1:** improve(T)

Q: priority queue of loose paths in T
  //ordered by decreasing weights

while Q.notEmpty() do
  p = Q.dequeue()
  \{T_1, T_2\} = Remove(p, T)
  findShortestPath(T_1, T_2)
  //shortest path between T_1 and T_2 in G

  if w(T') < w(T) then
    T = T'
    Q: priority queue of loose paths in T
    //ordered by decreasing weights
  end if
end while
return T
STAR: Phase II

- **Definitions:**
  1. **Fixed node:** either a query node or a node of degree >2 in the current tree
  2. **Loose path:** path of the current tree in which only end nodes are fixed nodes

**Algorithm 1: improve(T)**

- Q: priority queue of loose paths in T
  //ordered by decreasing weights

  while Q.notEmpty() do
    p = Q.dequeue()
    \( \{T_1, T_2\} = \text{Remove}(p, T) \)
    findShortestPath\( (T_1, T_2) \)
    //shortest path between \( T_1 \) and \( T_2 \) in \( G \)

    if \( w(T') < w(T) \) then
      T = T'
      Q: priority queue of loose paths in T
      //ordered by decreasing weights
    end if
  end while

return T
STAR: Phase II

- **Definitions:**
  1. **Fixed node:** either a query node or a node of degree >2 in the current tree
  2. **Loose path:** path of the current tree in which only end nodes are fixed nodes

**Algorithm 1: improve(T)**

Q: priority queue of *loose paths* in T  
//ordered by decreasing weights

**while** Q.notEmpty() **do**

p = Q.dequeue()

\( \{T_1, T_2\} = \text{Remove}(p, T) \)

findShortestPath(T_1, T_2)  
//shortest path between \( T_1 \) and \( T_2 \) in G

**if** \( w(T') < w(T) \) **then**

T = T'

Q: priority queue of *loose paths* in T  
//ordered by decreasing weights

**end if**

**end while**

**return** T
Example: Phase II

- entity
- subClassOf
- person
- subClassOf
- actor
- type
- state
- subClassOf
- scientist
- subClassOf
- politician
- type
- Max Planck Institute
- type
- locatedIn
- Germany
- namedAfter
- Max Planck
- chancellorOf
- Arnold Schwarzenegger
- type
- Angela Merkel
- type
- politician
- type
- scientist
- type
- Arnold Schwarzenegger
- type
Example: Phase II

- entity
- subClassOf
- person
- subClassOf
- actor
- type
- organization
- state
- type
- institute
- type
- state
- locatedIn
- Germany
- namedAfter
- Max Planck Institute
- Max Planck
- scientist
- subClassOf
- physicist
- type
- chancellorOf
- Arnold Schwarzenegger
- politician
- type
Example: Phase II
So what? …can’t we search for this tree right away?
Search space should be explored carefully!
STAR: Shortest Path Heuristic

Super fast construction of an initial tree

- Effective pruning of the local neighborhood (by choosing the longest loose path to replace)
- Only 2 SSSP iterators per improvement step
  ➔ Low cost for managing data structures
- Smart expansion strategy for iterators
  (Low-degree prioritization & Balanced expansion)

= Very efficient result generation

Algorithm 2 findShortestPath(V(T1), V(T2), lp)
1: for all \( v \in V \) do
2:   if \( v \in V(T_1) \) then \( d_1(v) = 0 \) else \( d_1(v) = \infty \)
3:   if \( v \in V(T_2) \) then \( d_2(v) = 0 \) else \( d_2(v) = \infty \)
4: end for
5: PriorityQueue \( Q_1 = V(T_1) \) //ordered by inc. distance \( d_1 \)
6: PriorityQueue \( Q_2 = V(T_2) \) //ordered by inc. distance \( d_2 \)
7: current = 1
8: other = 2
9: repeat
10: if fringe(Qother) < fringe(Qcurrent) then
11:   swap(current, other)
12: end if
13: \( v = Q_{current}.dequeue() \)
14: if \( d_{current}(v) \geq w(lp) \) then
15:   break
16: end if
17: for all \((v, v') \in E\) do
18:   if \( v' \) has been dequeued from \( Q_{current} \) then
19:     continue
20: end if
21: if \( d_{current}(v') > d_{current}(v) + w(v, v') \) then
22:   \( d_{current}(v') = d_{current}(v) + w(v, v') \)
23:   \( v'.predecessor_{current} = v \)
24: end if
25: \( Q_{current}.enqueue(v') \)
26: end for
27: until \( Q_1 = \emptyset \lor Q_2 = \emptyset \lor v \in V(T_{other}) \)
28: return path connecting \( T_1 \) and \( T_2 \)
STAR: Analysis

**Theorem 1**: For $l$ query entities, STAR yields an $O(\log l)$ approximation, independent of the initial tree size.
STAR: Analysis

**Theorem 1:** For $l$ query entities, STAR yields an $O(\log l)$ approximation, independent of the initial tree size.

Do not bother about the size of the first tree. Just get it as quickly as possible.
STAR: Analysis

**Theorem 1**: For \( l \) query entities, STAR yields an \( O(\log l) \) approximation, independent of the initial tree size.

Do not bother about the size of the first tree. Just get it as quickly as possible.

**Theorem 2**: STAR has a pseudo-polynomial run-time guarantee.
Theorem 1: For $l$ query entities, STAR yields an $O(\log l)$ approximation, independent of the initial tree size.

Do not bother about the size of the first tree. Just get it as quickly as possible.

Theorem 2: STAR has a pseudo-polynomial run-time guarantee.

... in theory, and very efficient in practice.
Algorithm 3: getTopK(T, k)  // T being the result of phase II

Q: priority queue of trees
// generated during the improvement process of phase II
// ordered by decreasing weights

while Q.size < k do
    T' = improve'(relaxWeights(T, ε))
    // T cannot be locally improved unless
    // its edge weights are artificially relaxed
    // improve' guarantees node-disjoint improvement

    T = reweight(T')
    // assigns original weights

    Q.enqueue(T)
end while

return T

All trees produced during the improvement process are stored in the priority queue Q

→ Number of trees in Q grows quickly during the improvement process
Outline

✓ Intro & Related Work
✓ STAR:
  ✓ Algorithm & Heuristics
  ✓ Analysis
  ✓ Top-k
• Experiments
• Conclusion
Experiments

• Efficiency oriented approaches
  BANKS I [Bhalotia et al. ICDE’02],
  BANKS II [Kacholia et al. VLDB’05]
  BLINKS [He et al. SIGMOD’07]

• Approximation oriented approaches
  DPBF [Ding et al. ICDE’07],
  DNH [Kou et al. AI 1981]

Main mem. top-1 comparison on DBLP (15K N, 150K E)
(60 random queries for each number of query entities)

<table>
<thead>
<tr>
<th>Method</th>
<th># query entities</th>
<th>Avg. weight</th>
<th>Avg. runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>STAR</td>
<td>3</td>
<td>0.61</td>
<td>604.2</td>
</tr>
<tr>
<td>DNH</td>
<td></td>
<td>0.7</td>
<td>5402.9</td>
</tr>
<tr>
<td>DPBF</td>
<td></td>
<td>0.58</td>
<td>33096.7</td>
</tr>
<tr>
<td>BANKS I</td>
<td></td>
<td>1.22</td>
<td>2096.3</td>
</tr>
<tr>
<td>BANKS II</td>
<td></td>
<td>1.81</td>
<td>3214.1</td>
</tr>
<tr>
<td>STAR</td>
<td>5</td>
<td>0.86</td>
<td>960.2</td>
</tr>
<tr>
<td>DNH</td>
<td></td>
<td>0.98</td>
<td>9166.7</td>
</tr>
<tr>
<td>DPBF</td>
<td></td>
<td>0.81</td>
<td>432361.5</td>
</tr>
<tr>
<td>BANKS I</td>
<td></td>
<td>1.87</td>
<td>3617.3</td>
</tr>
<tr>
<td>BANKS II</td>
<td></td>
<td>2.46</td>
<td>5797.5</td>
</tr>
<tr>
<td>STAR</td>
<td>7</td>
<td>1.12</td>
<td>1579.6</td>
</tr>
<tr>
<td>DNH</td>
<td></td>
<td>1.22</td>
<td>17430.9</td>
</tr>
<tr>
<td>DPBF</td>
<td></td>
<td>?</td>
<td>5945.5</td>
</tr>
<tr>
<td>BANKS I</td>
<td></td>
<td>2.37</td>
<td>9435.5</td>
</tr>
<tr>
<td>BANKS II</td>
<td></td>
<td>3.42</td>
<td></td>
</tr>
</tbody>
</table>
Experiments

• Efficiency oriented approaches
  BANKS I [Bhalotia et al. ICDE’02],
  BANKS II [Kacholia et al. VLDB’05]
  BLINKS [He et al. SIGMOD’07]

• Approximation oriented approaches
  DPBF [Ding et al. ICDE’07],
  DNH [Kou et al. AI 1981]

<table>
<thead>
<tr>
<th>Method</th>
<th>Top-k</th>
<th>Avg. weight</th>
<th>Avg. runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>STAR</td>
<td>10</td>
<td>1.57</td>
<td>1206.3</td>
</tr>
<tr>
<td>BANKS I</td>
<td></td>
<td>2.43</td>
<td>5851.8</td>
</tr>
<tr>
<td>BANKS II</td>
<td></td>
<td>3.78</td>
<td>7895.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>n/a</td>
<td>19051.4</td>
</tr>
<tr>
<td>STAR</td>
<td>50</td>
<td>2.23</td>
<td>3118.3</td>
</tr>
<tr>
<td>BANKS I</td>
<td></td>
<td>3.12</td>
<td>7335.1</td>
</tr>
<tr>
<td>BANKS II</td>
<td></td>
<td>5.31</td>
<td>8928.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>n/a</td>
<td>21837.9</td>
</tr>
<tr>
<td>STAR</td>
<td>100</td>
<td>3.01</td>
<td>4705.1</td>
</tr>
<tr>
<td>BANKS I</td>
<td></td>
<td>4.51</td>
<td>9640.8</td>
</tr>
<tr>
<td>BANKS II</td>
<td></td>
<td>6.81</td>
<td>11071.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>n/a</td>
<td>24632.3</td>
</tr>
</tbody>
</table>

Main mem. top-k comparison on DBLP (15K N, 150K E)
(60 random queries for each k; 5 query entities per query)

<table>
<thead>
<tr>
<th>Method</th>
<th># query entities</th>
<th>Avg. weight</th>
<th>Avg. runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>STAR</td>
<td>3</td>
<td>0.61</td>
<td>604.2</td>
</tr>
<tr>
<td>DNH</td>
<td></td>
<td>0.7</td>
<td>5402.9</td>
</tr>
<tr>
<td>DPBF</td>
<td></td>
<td>0.58</td>
<td>33096.7</td>
</tr>
<tr>
<td>BANKS I</td>
<td></td>
<td>1.22</td>
<td>2096.3</td>
</tr>
<tr>
<td>BANKS II</td>
<td></td>
<td>1.81</td>
<td>3214.1</td>
</tr>
<tr>
<td>STAR</td>
<td>5</td>
<td>0.86</td>
<td>960.2</td>
</tr>
<tr>
<td>DNH</td>
<td></td>
<td>0.98</td>
<td>9166.7</td>
</tr>
<tr>
<td>DPBF</td>
<td></td>
<td>0.81</td>
<td>432361.5</td>
</tr>
<tr>
<td>BANKS I</td>
<td></td>
<td>1.87</td>
<td>3617.3</td>
</tr>
<tr>
<td>BANKS II</td>
<td></td>
<td>2.46</td>
<td>5797.5</td>
</tr>
<tr>
<td>STAR</td>
<td>7</td>
<td>1.12</td>
<td>1579.6</td>
</tr>
<tr>
<td>DNH</td>
<td></td>
<td>1.22</td>
<td>17430.9</td>
</tr>
<tr>
<td>DPBF</td>
<td></td>
<td>2.37</td>
<td>5945.5</td>
</tr>
<tr>
<td>BANKS I</td>
<td></td>
<td>3.42</td>
<td>9435.5</td>
</tr>
</tbody>
</table>

Main mem. top-1 comparison on DBLP (15K N, 150K E)
(60 random queries for each number of query entities)
Conclusion

Super fast construction of an initial tree
(don’t care about its weight)
Conclusion

Super fast construction of an initial tree
(don’t care about its weight)

+ Fast local search by effectively pruning the
local neighborhood of the current tree
(choose always the longest loose path to replace)
Conclusion

- Super fast construction of an initial tree (don’t care about its weight)

- Fast local search by effectively pruning the local neighborhood of the current tree (choose always the longest loose path to replace)

- Low cost for managing data structures (only 2 SSSP iterators per improvement step)
Conclusion

Super fast construction of an initial tree
(don’t care about its weight)

+ Fast local search by effectively pruning the
  local neighborhood of the current tree
  (choose always the longest loose path to replace)

+ Low cost for managing data structures
  (only 2 SSSP iterators per improvement step)

+ Smart exploration of the search space
  (low-degree prioritization & balanced expansion)
Conclusion

Super fast construction of an initial tree
(don’t care about its weight)

+ Fast local search by effectively pruning the
  local neighborhood of the current tree
  (choose always the longest loose path to replace)

+ Low cost for managing data structures
  (only 2 SSSP iterators per improvement step)

+ Smart exploration of the search space
  (low-degree prioritization & balanced expansion)

+ Almost no waste
  (every improvement leads to a top-k candidate)
Conclusion

Super fast construction of an initial tree
(don’t care about its weight)

+ Fast local search by effectively pruning the
  local neighborhood of the current tree
  (choose always the longest loose path to replace)

+ Low cost for managing data structures
  (only 2 SSSP iterators per improvement step)

+ Smart exploration of the search space
  (low-degree prioritization & balanced expansion)

+ Almost no waste
  (every improvement leads to a top-k candidate)

= Very efficient generation of top-k results
Conclusion

Super fast construction of an initial tree
(don’t care about its weight)

+ Fast local search by effectively pruning the
  local neighborhood of the current tree
  (choose always the longest loose path to replace)

+ Low cost for managing data structures
  (only 2 SSSP iterators per improvement step)

+ Smart exploration of the search space
  (low-degree prioritization & balanced expansion)

+ Almost no waste
  (every improvement leads to a top-k candidate)

= Very efficient generation of top-k results

Implemented as a query answering component of NAGA
www.mpii.de/~kasneci/naga
Outline

✓ Intro & Related Work
✓ STAR:
  ✓ Algorithm & Heuristics
  ✓ Analysis
  ✓ Top-k
✓ Experiments
✓ Conclusion