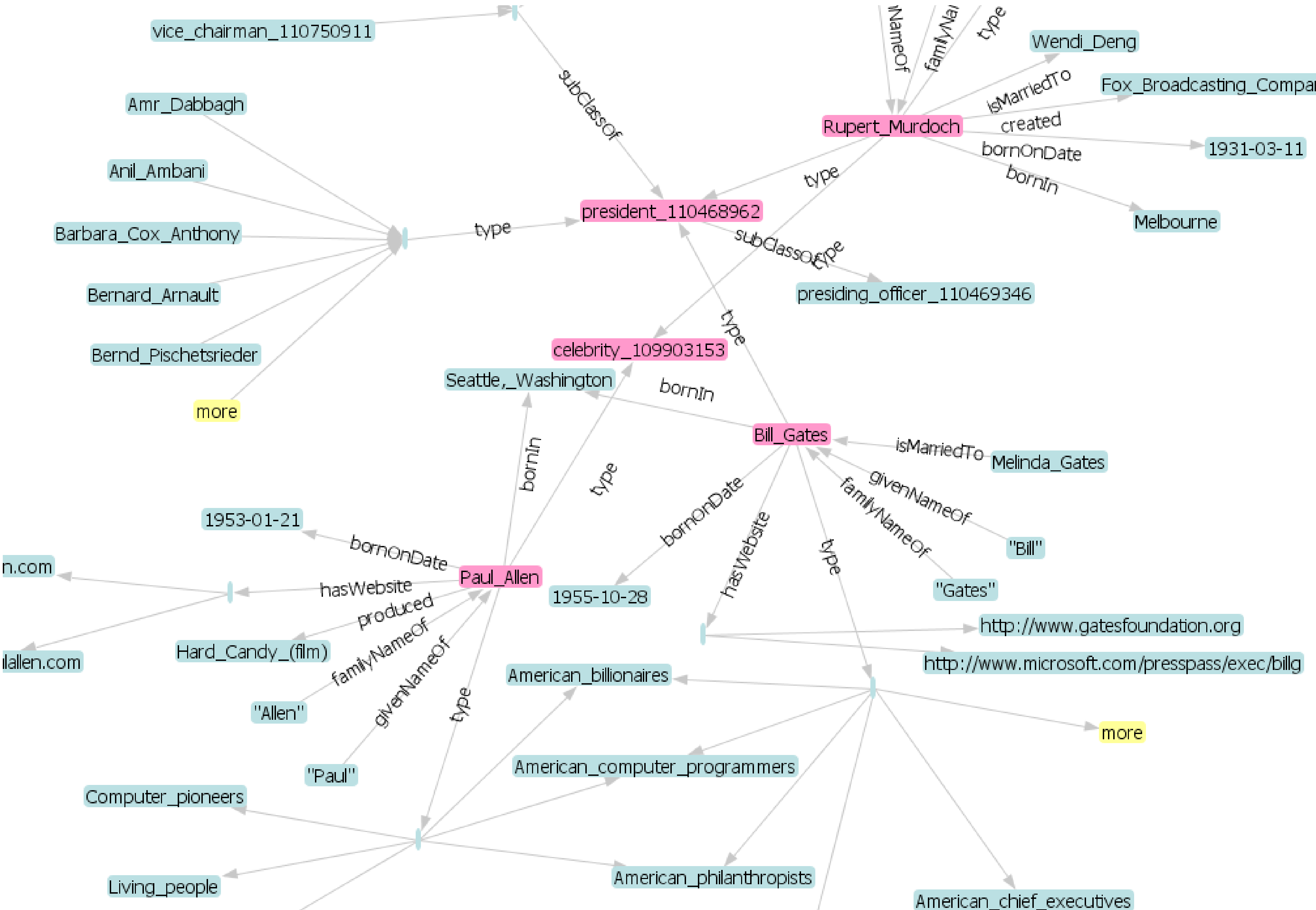
STAR: Steiner Tree Approximation in Relationship Graphs

Gjergji Kasneci

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

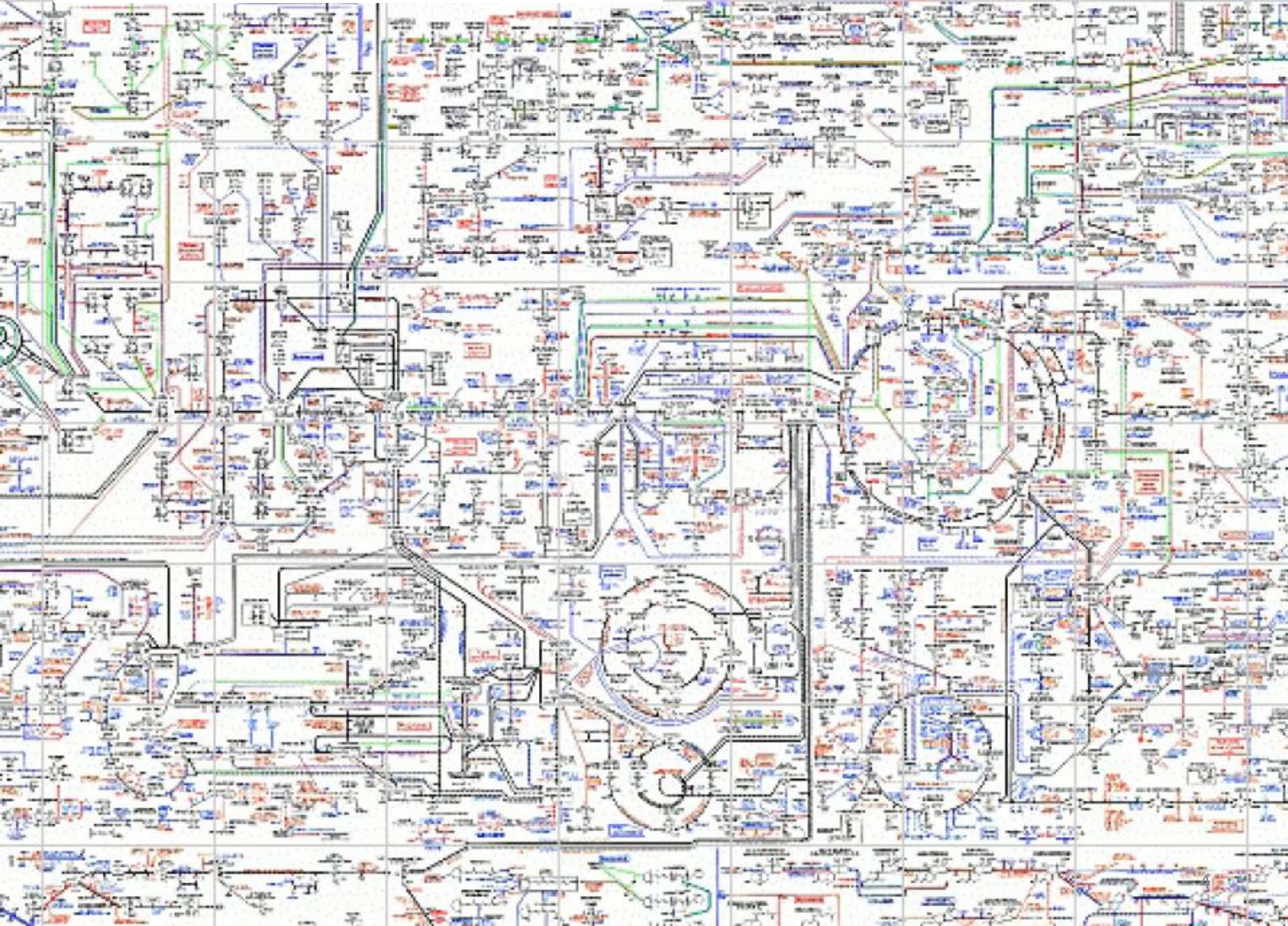
Relationship Graphs

- 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, …)
  - …



Excerpt from 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

  - Find top-k min-cost subtrees that interconnect all query nodes

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]

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]

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]

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

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

n : # nodes in G m : # edges in G l : # query terms 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)

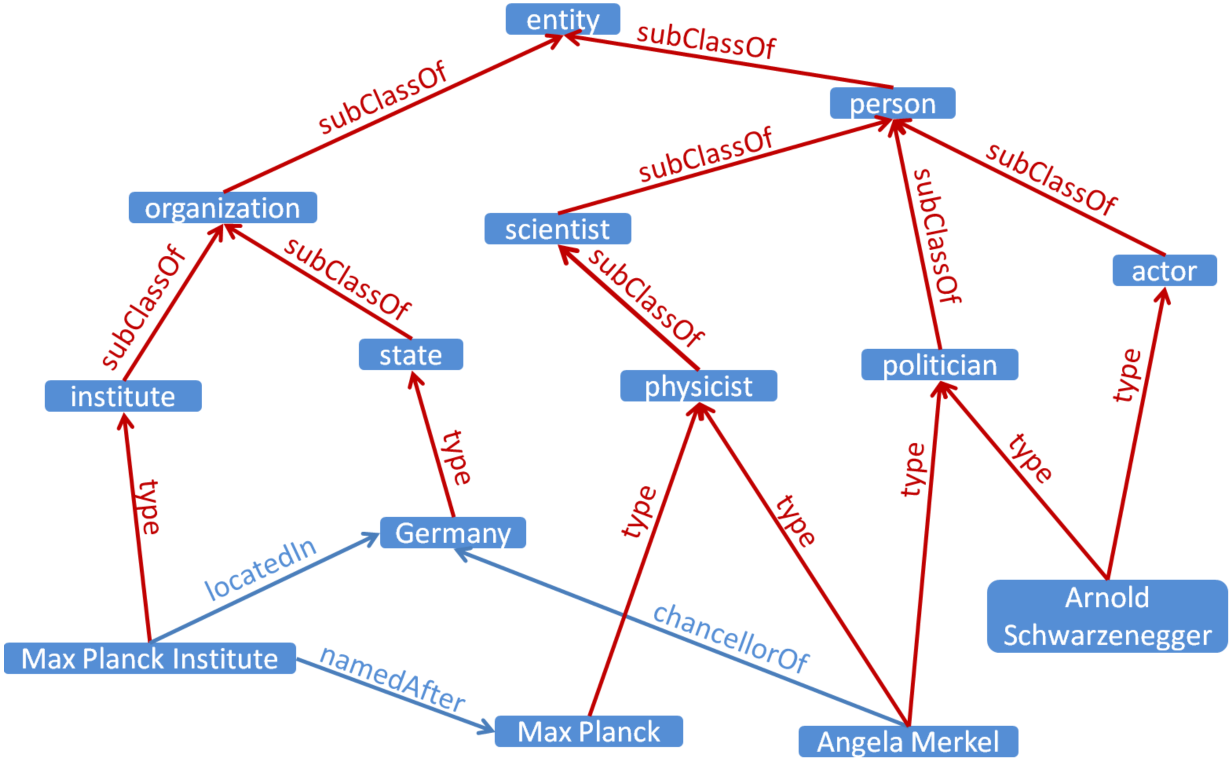
STAR: A Metaheuristic

- 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

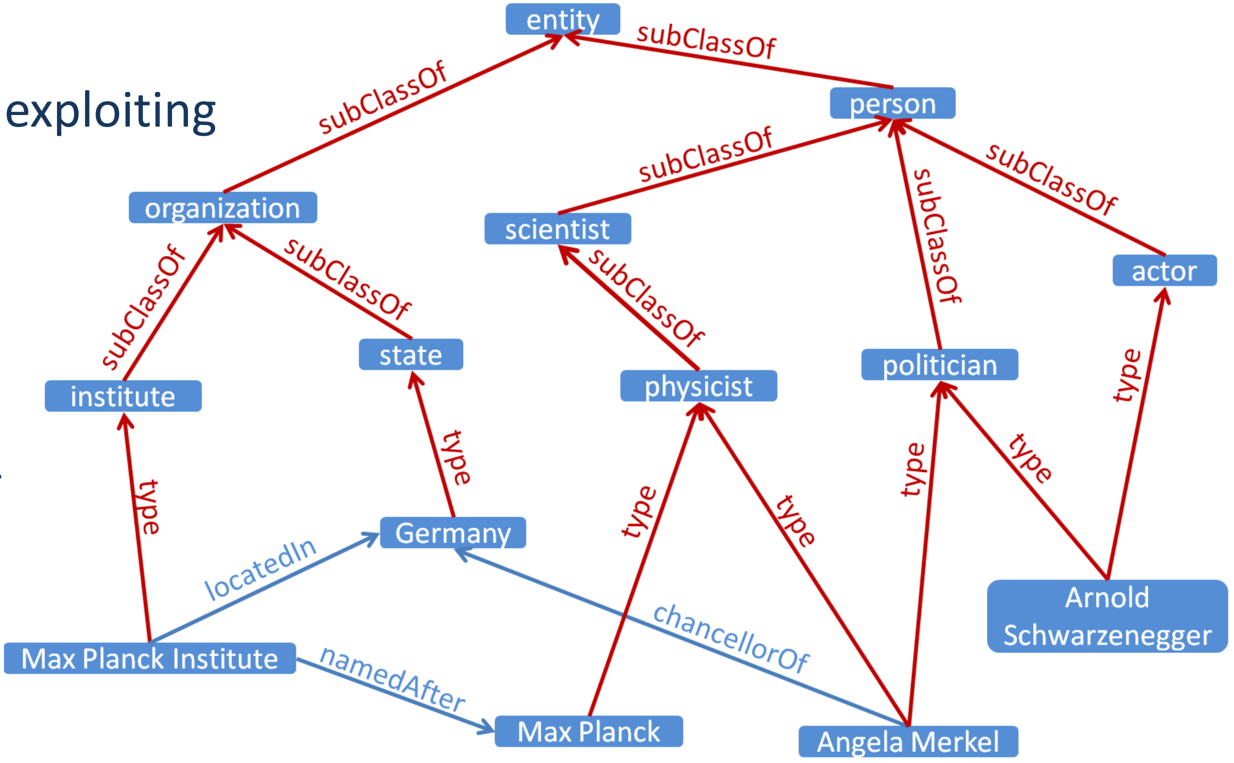
STAR: Phase I

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



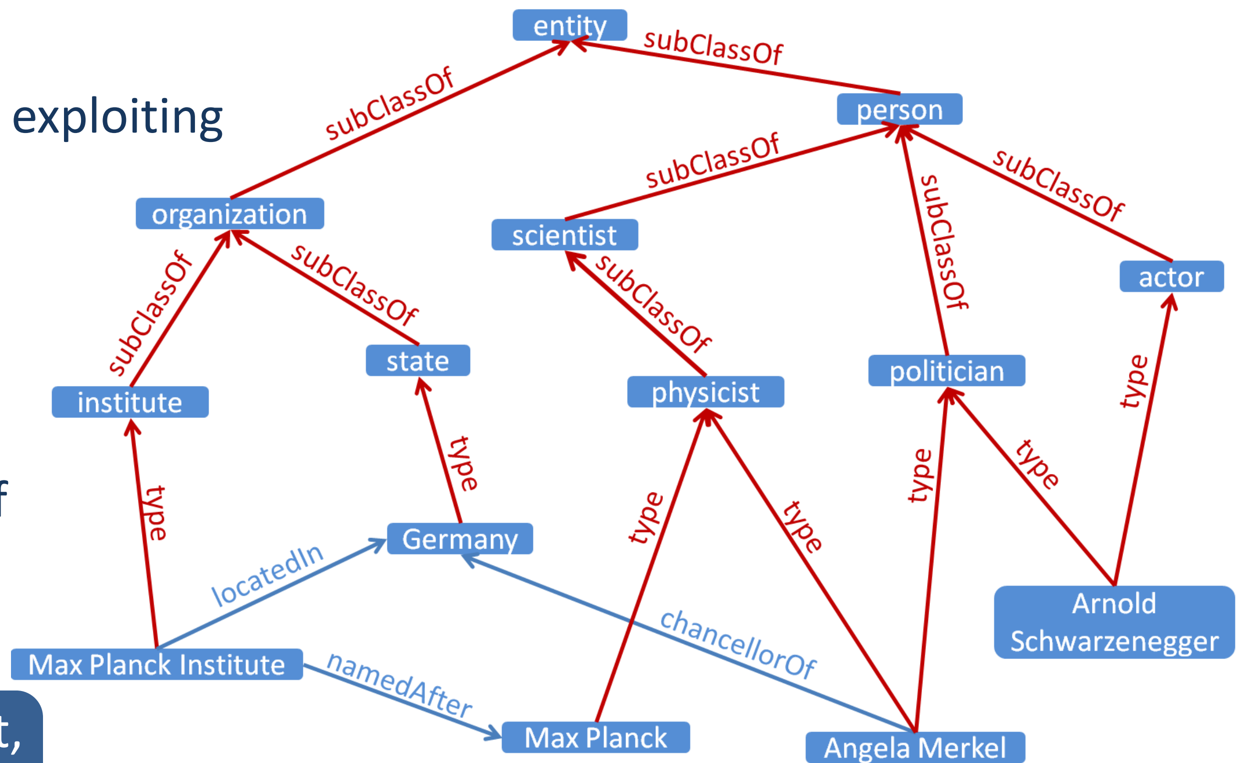
STAR: Phase I

- 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



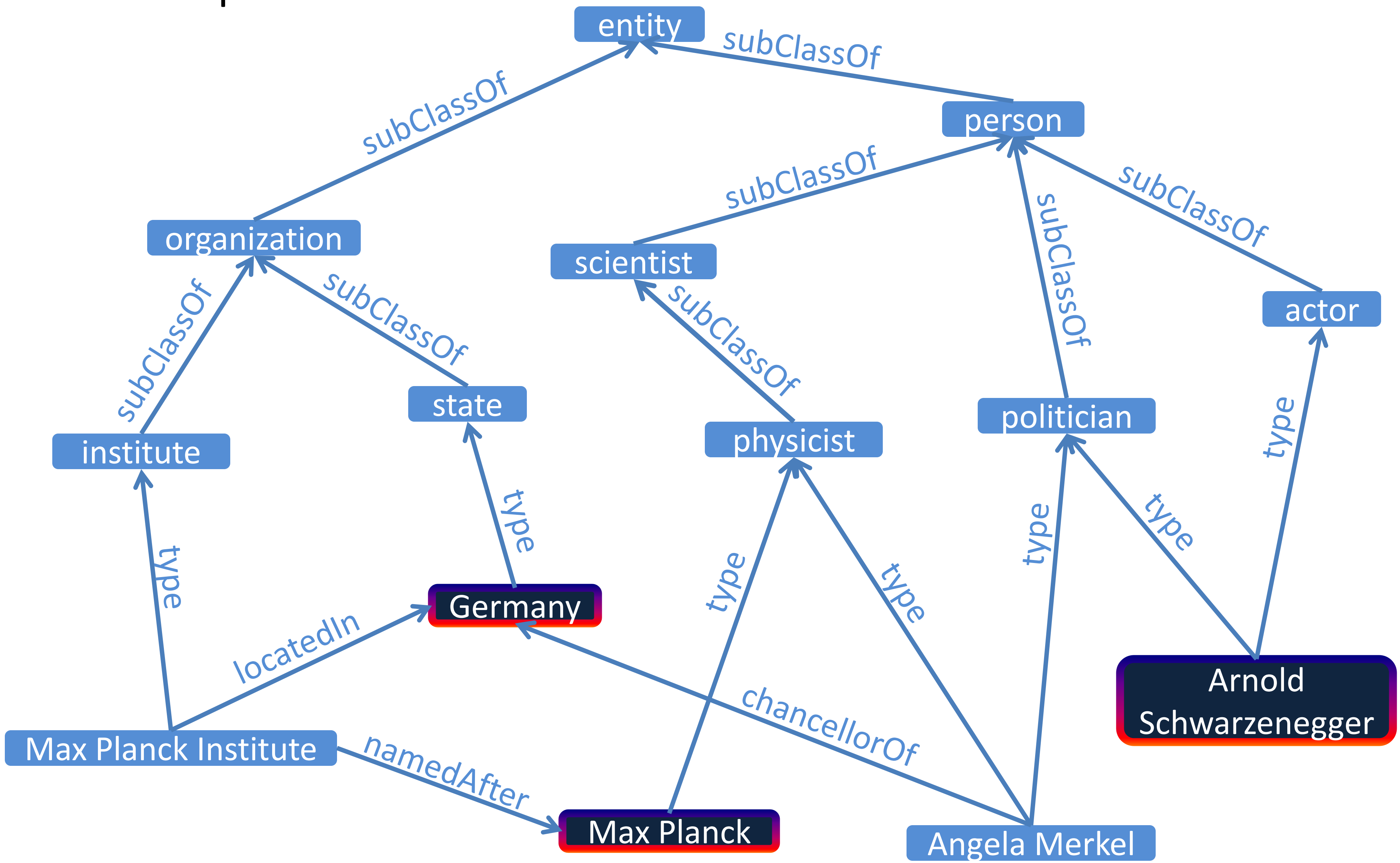
STAR: Phase I

- 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

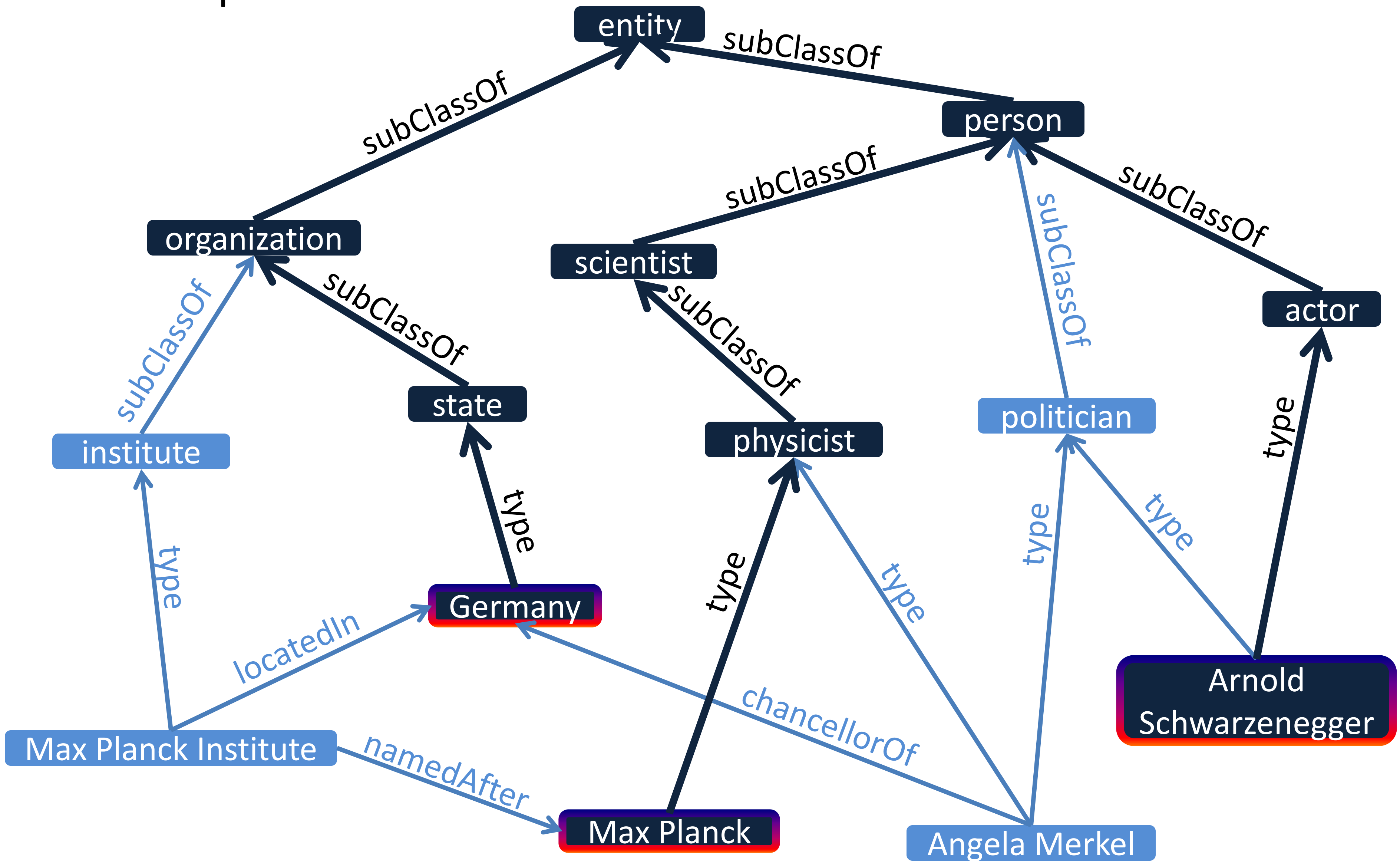


→ Very few edges to visit,
→ Very efficient

Example: Phase I



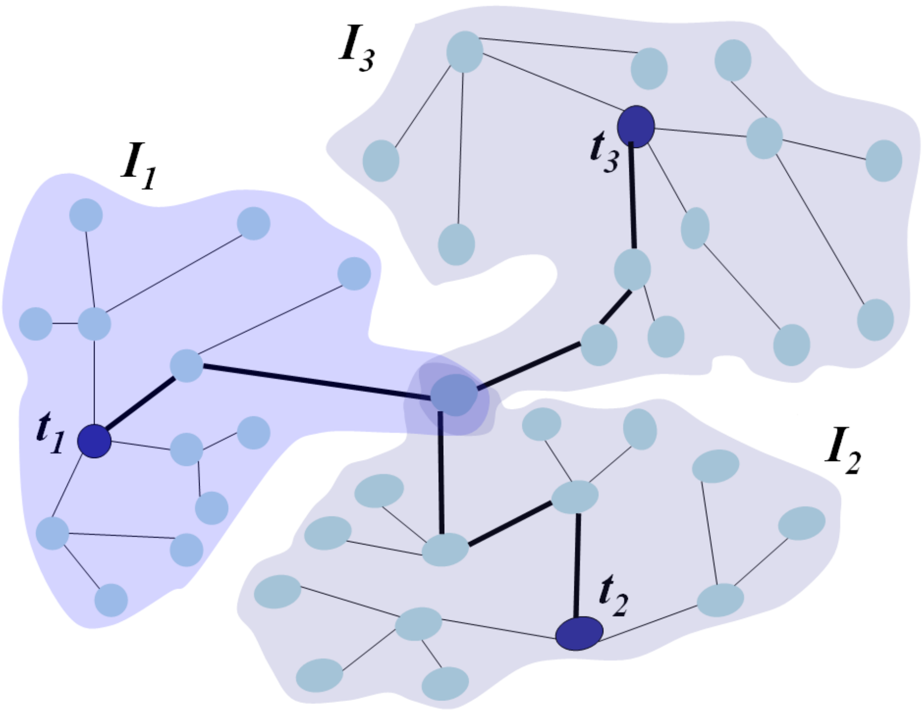
Example: Phase I



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()
      {T1, T2} = Remove(p, T)
      findShortestPath(T1, T2)
      //shortest path between T1 and T2 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()
      {T1, T2} = Remove(p, T)
      findShortestPath(T1, T2)
      //shortest path between T1 and T2 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

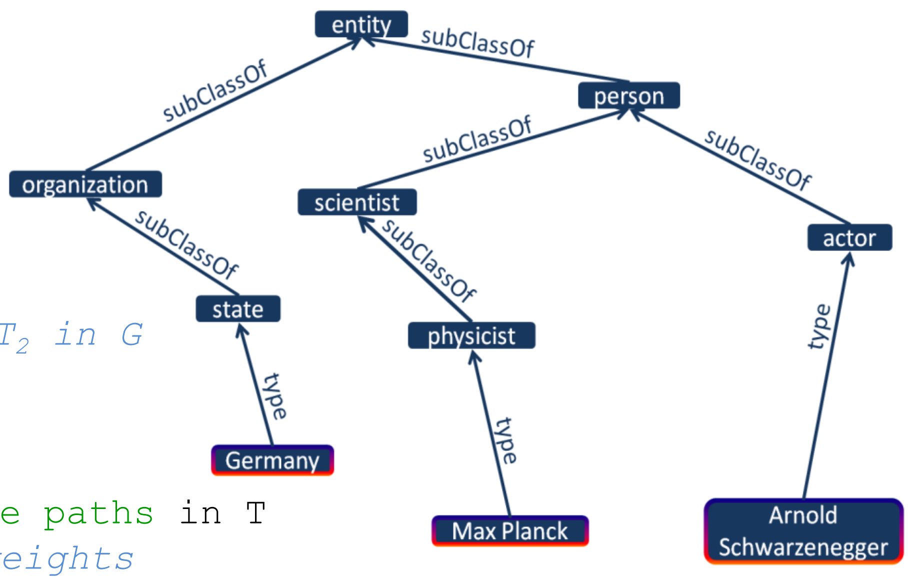
- **Definitions:**

  (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()
      {T1, T2} = Remove(p, T)
      findShortestPath(T1, T2)
      //shortest path between T1 and T2 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

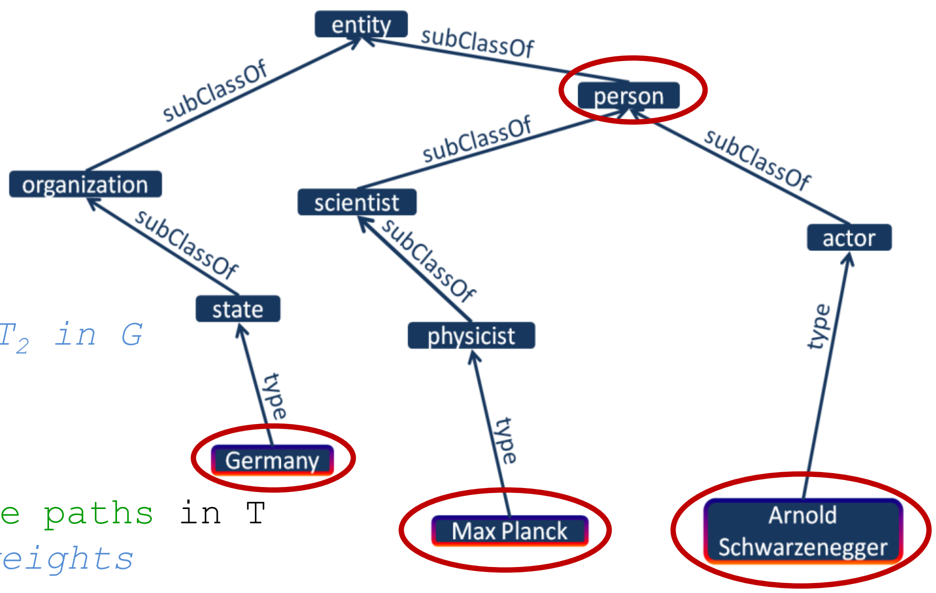
- **Definitions:**

  (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()
      {T1, T2} = Remove(p, T)
      findShortestPath(T1, T2)
      //shortest path between T1 and T2 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
```

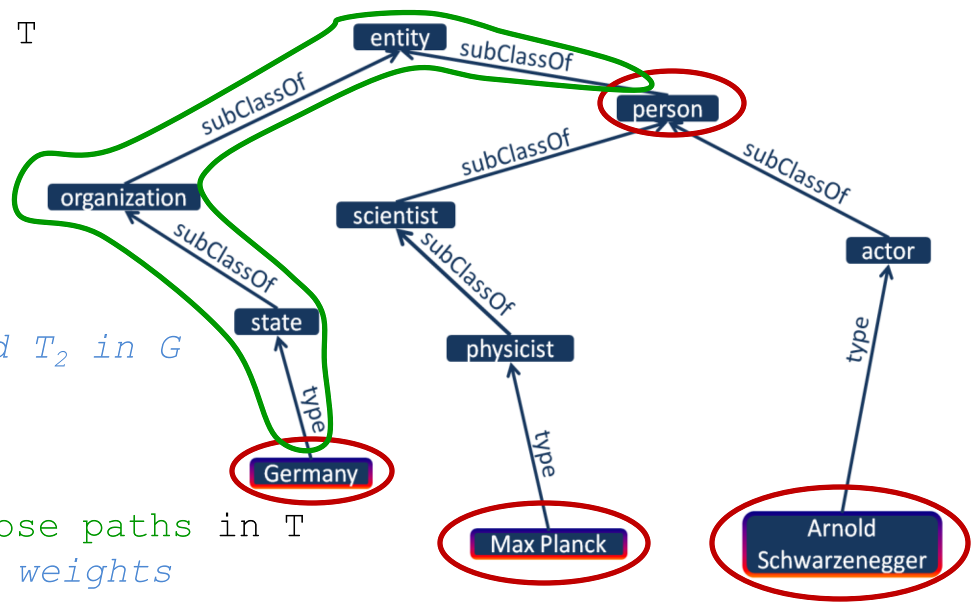


STAR: Phase II

- **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()
      {T1, T2} = Remove(p, T)
      findShortestPath(T1, T2)
      //shortest path between T1 and T2 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

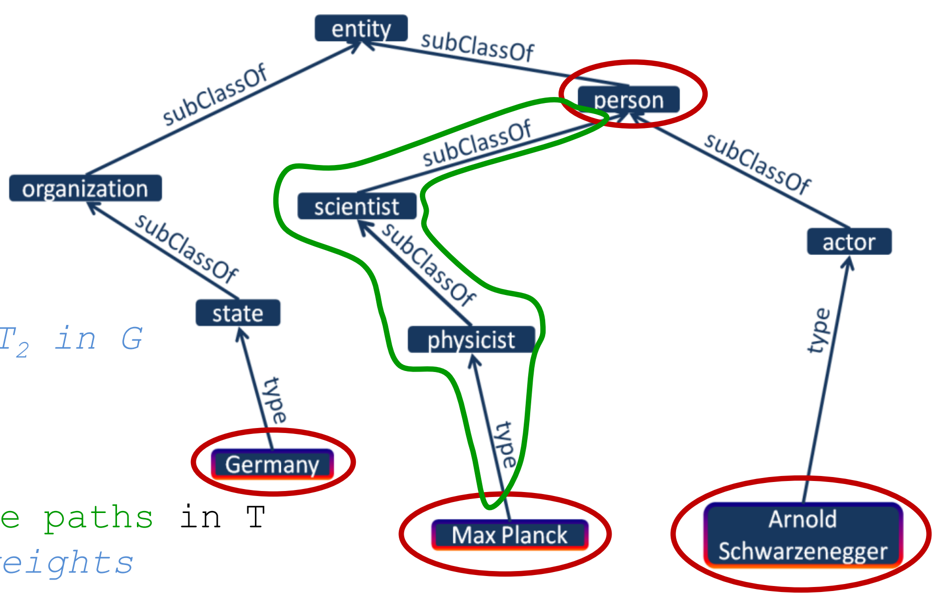
- **Definitions:**

  (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()
      {T1, T2} = Remove(p, T)
      findShortestPath(T1, T2)
      //shortest path between T1 and T2 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
```

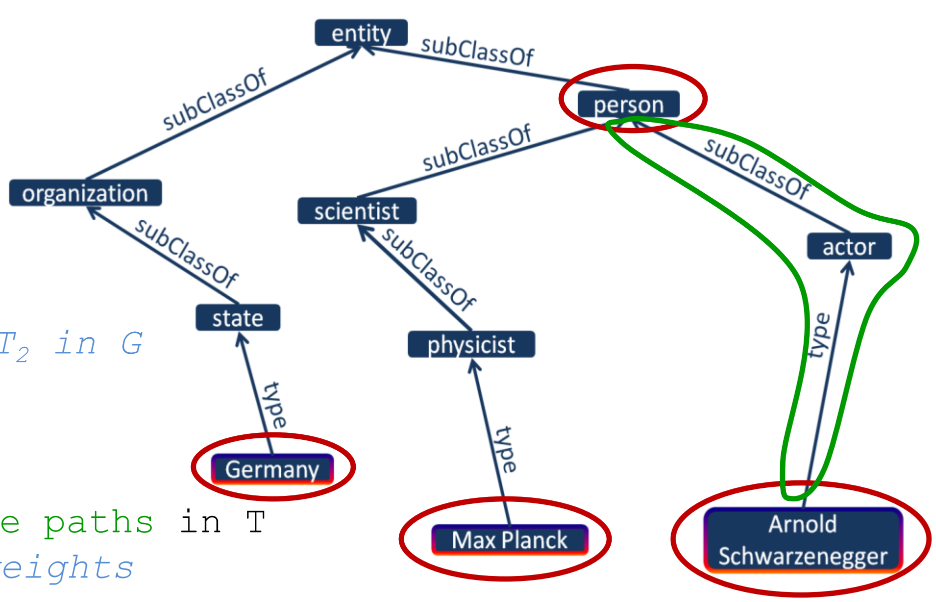


STAR: Phase II

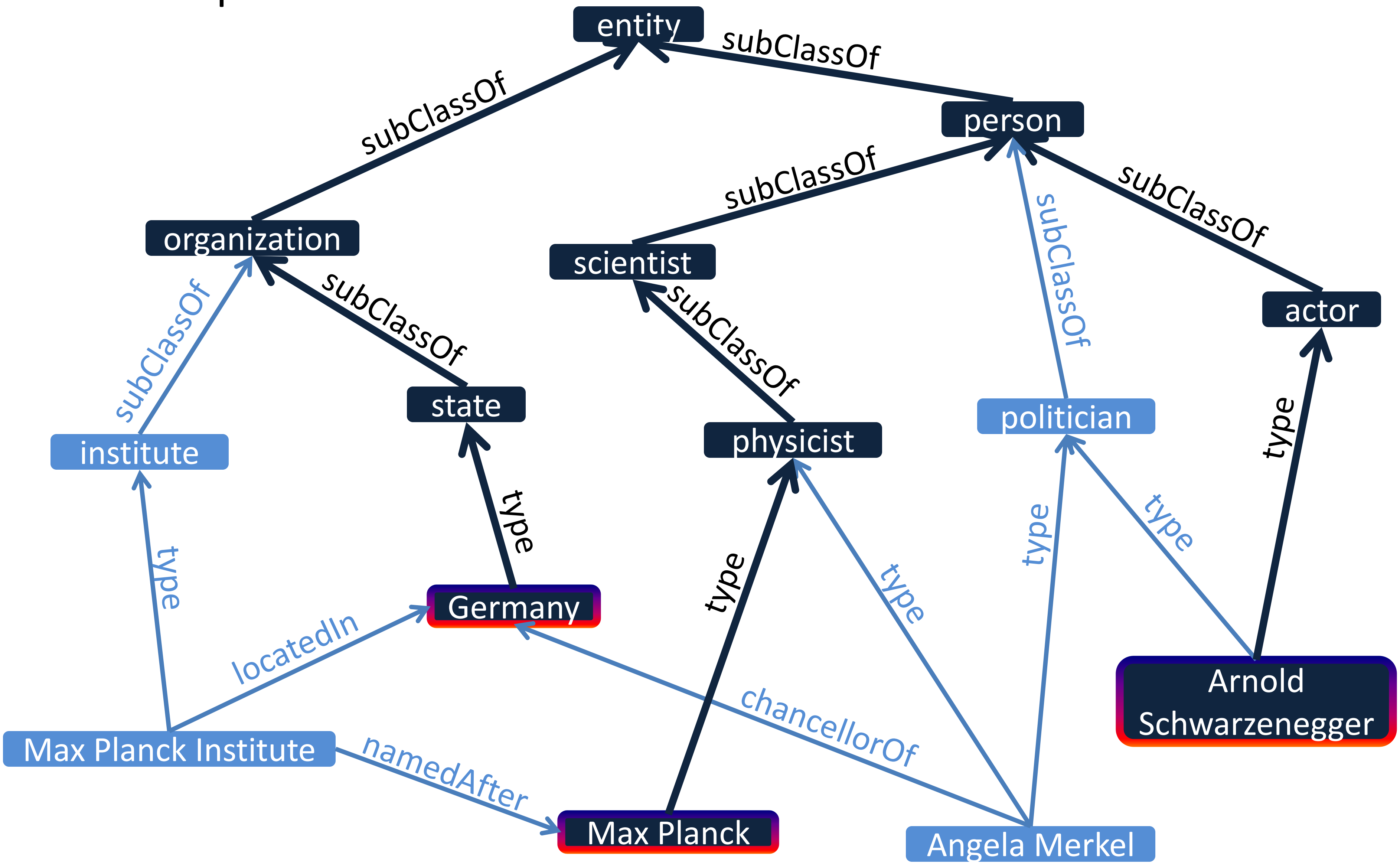
- **Definitions:**
  (1) Fixed node: *either a query node or a node of degree >2 in the current tree*
  (2) Loose path: *path of the current tree in which only end nodes are fixed nodes*

Algorithm 1: improve(T)

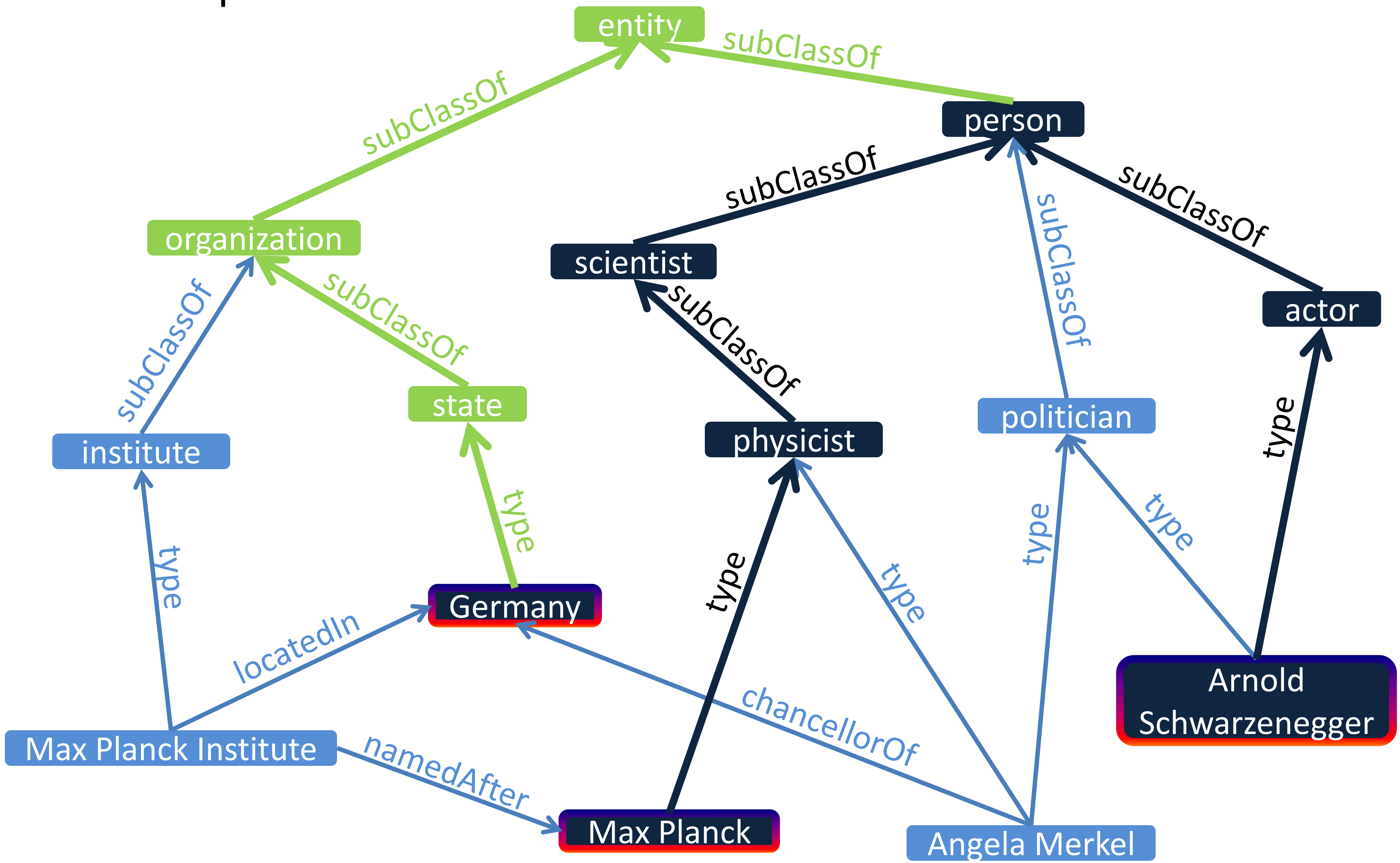
```
Q: priority queue of loose paths in T
//ordered by decreasing weights

while  Q.notEmpty()  do
      p = Q.dequeue()
      {T1, T2} = Remove(p, T)
      findShortestPath(T1, T2)
      //shortest path between T1 and T2 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
```



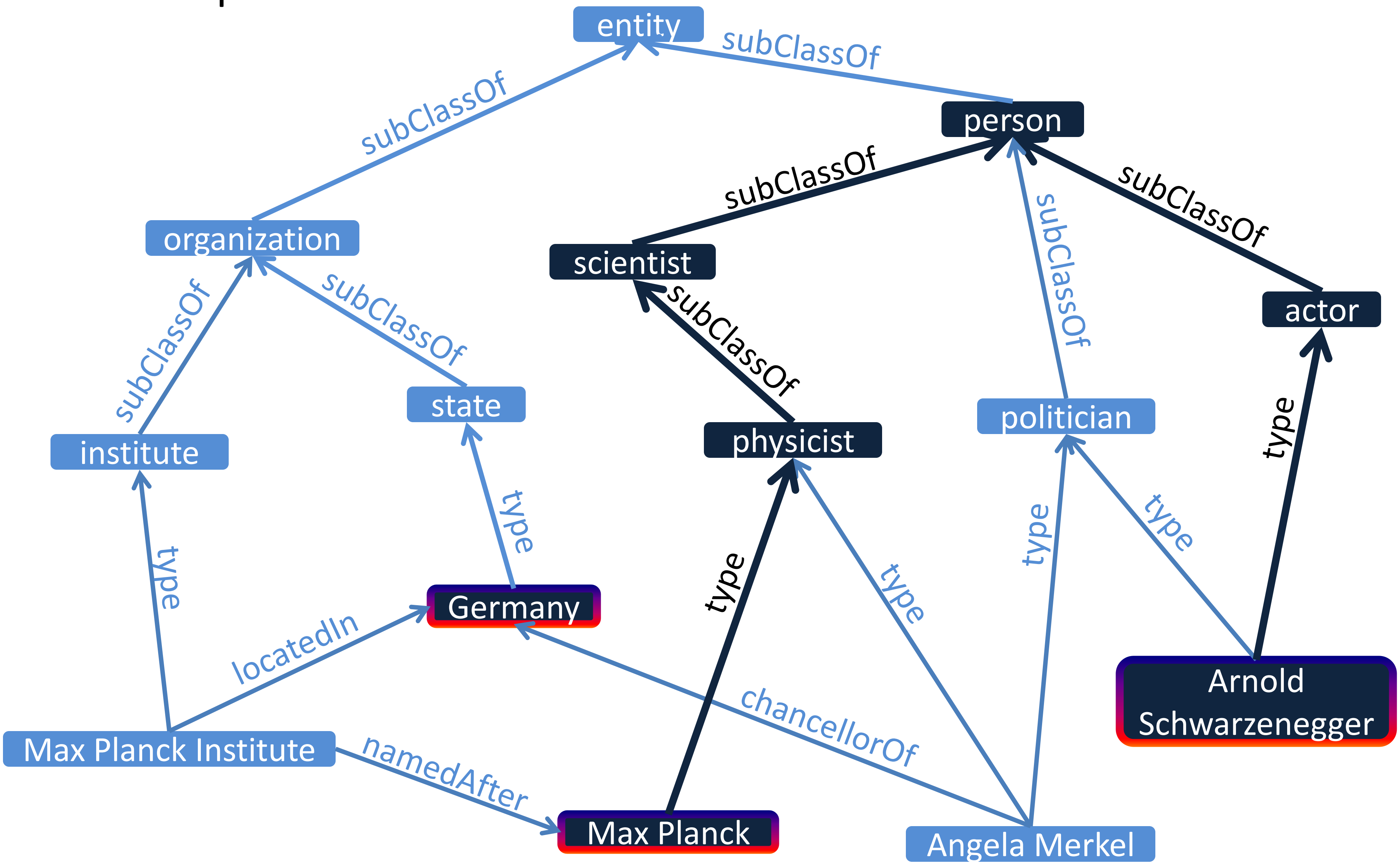
Example: Phase II



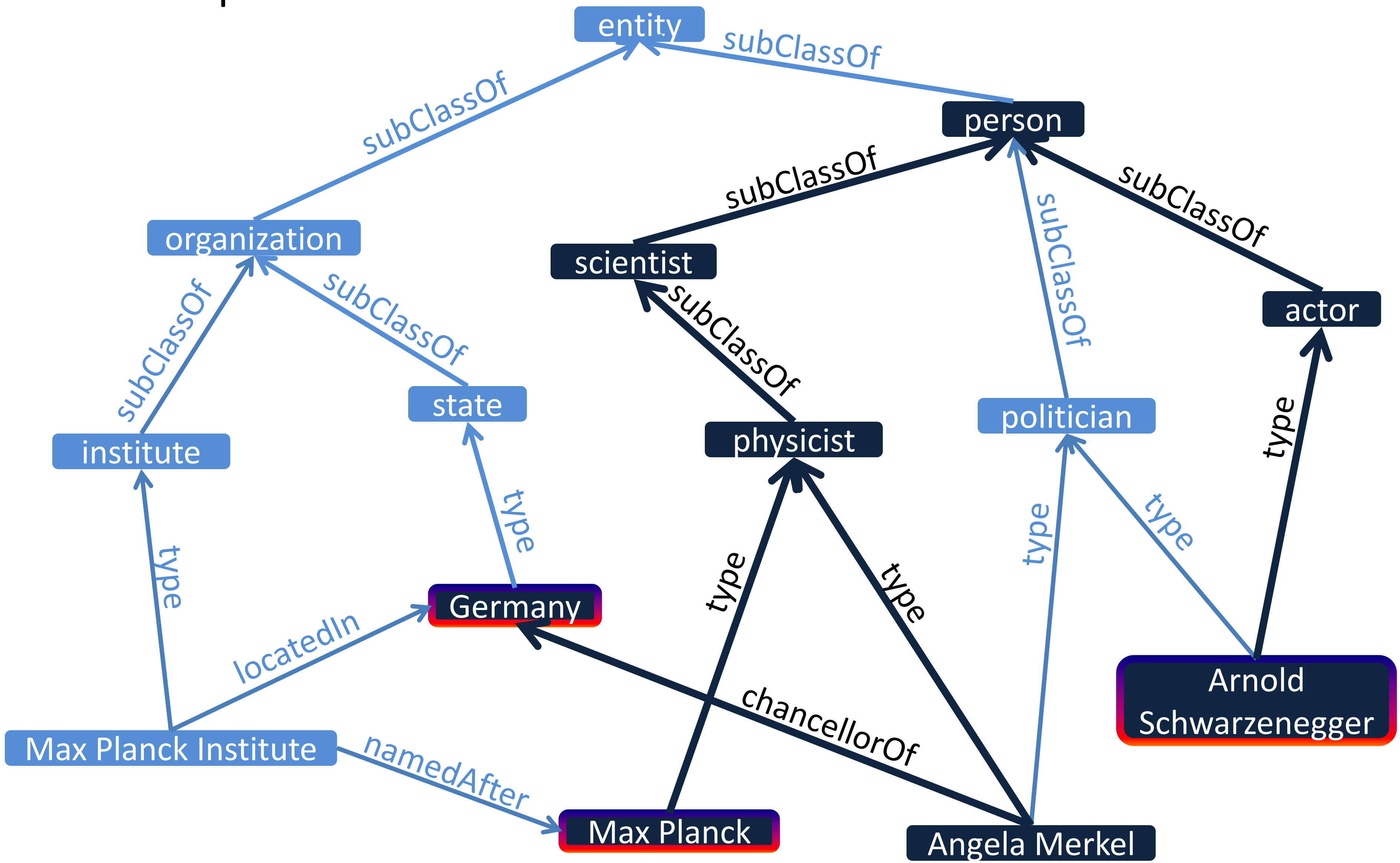
Example: Phase II



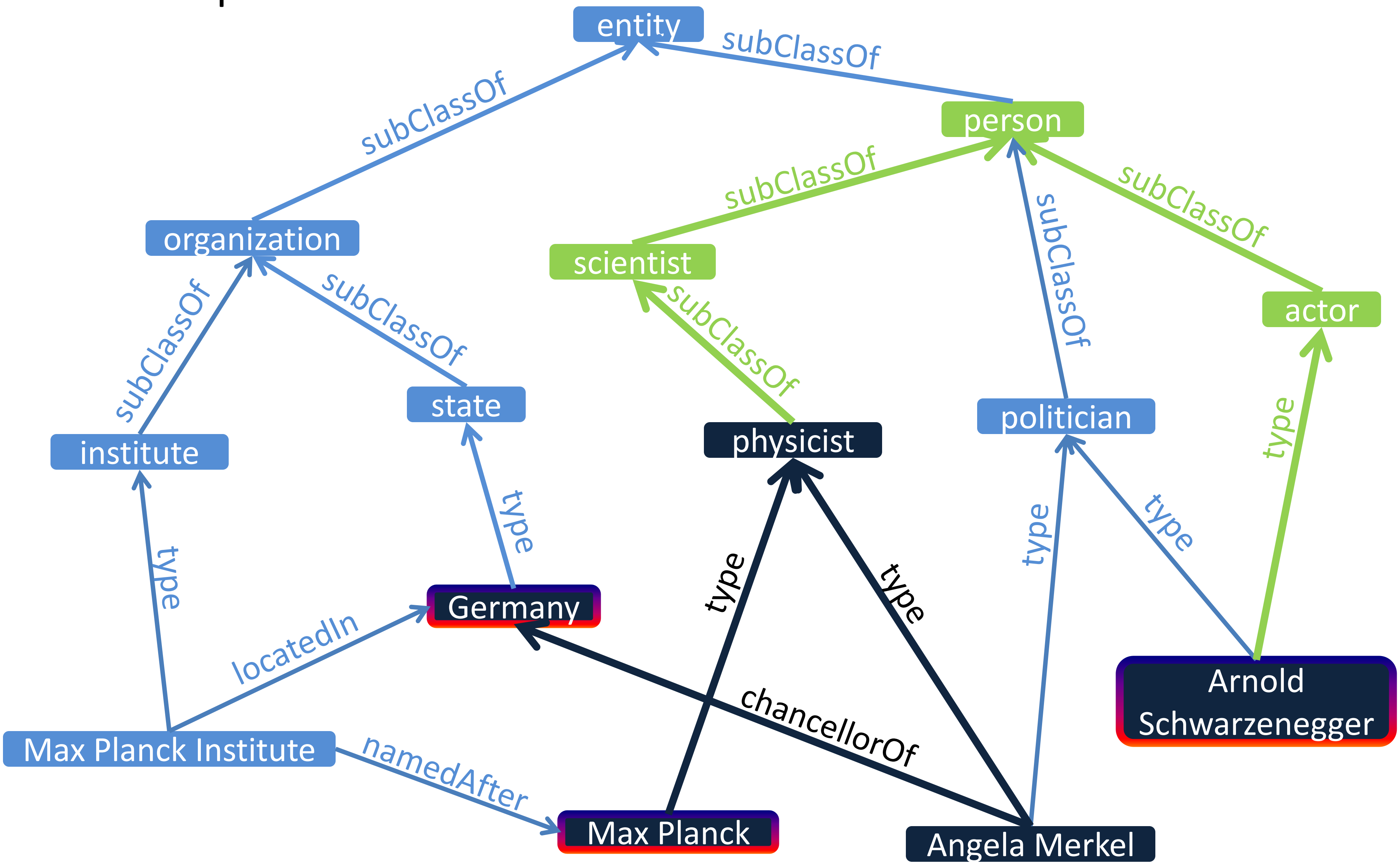
Example: Phase II



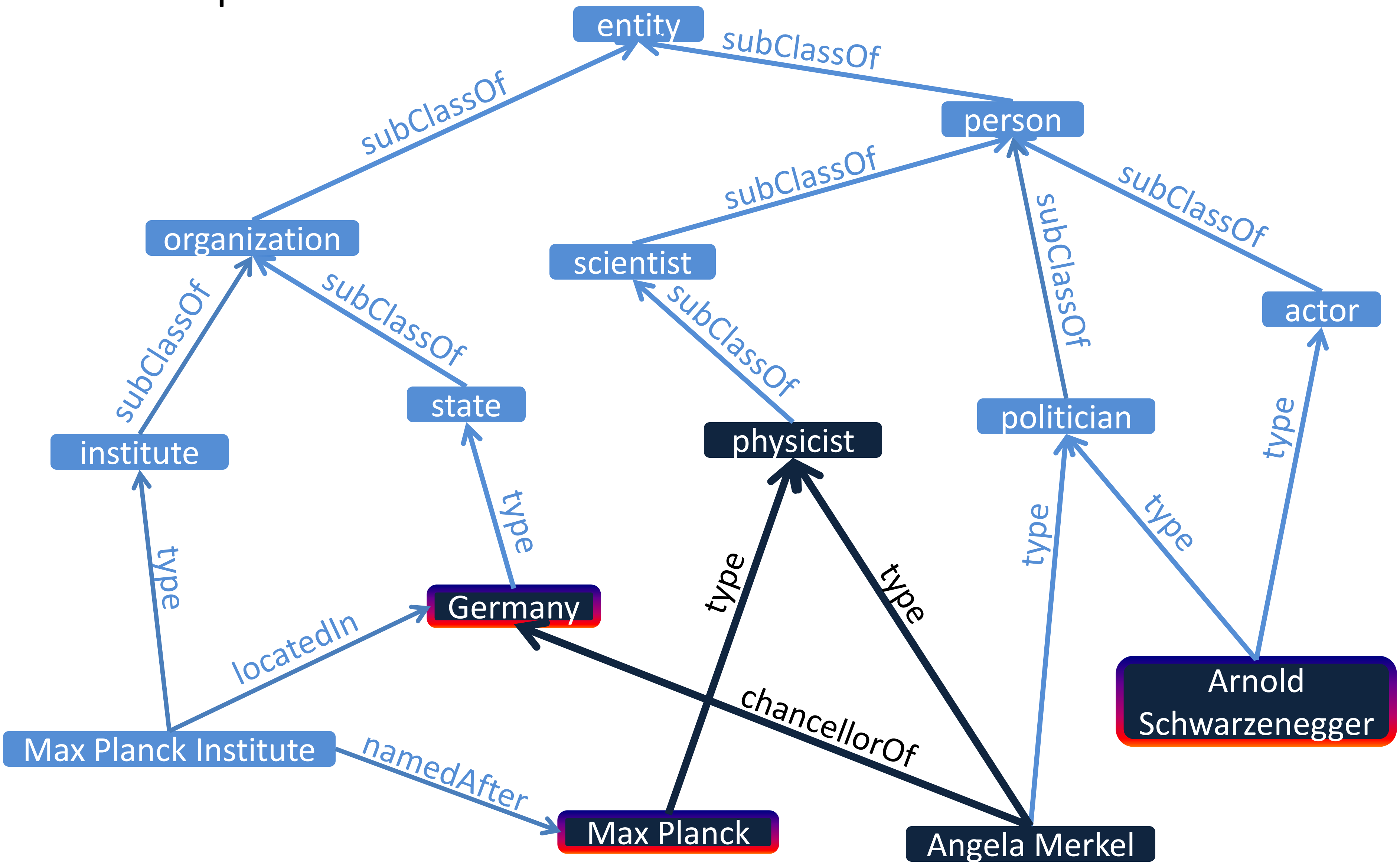
Example: Phase II



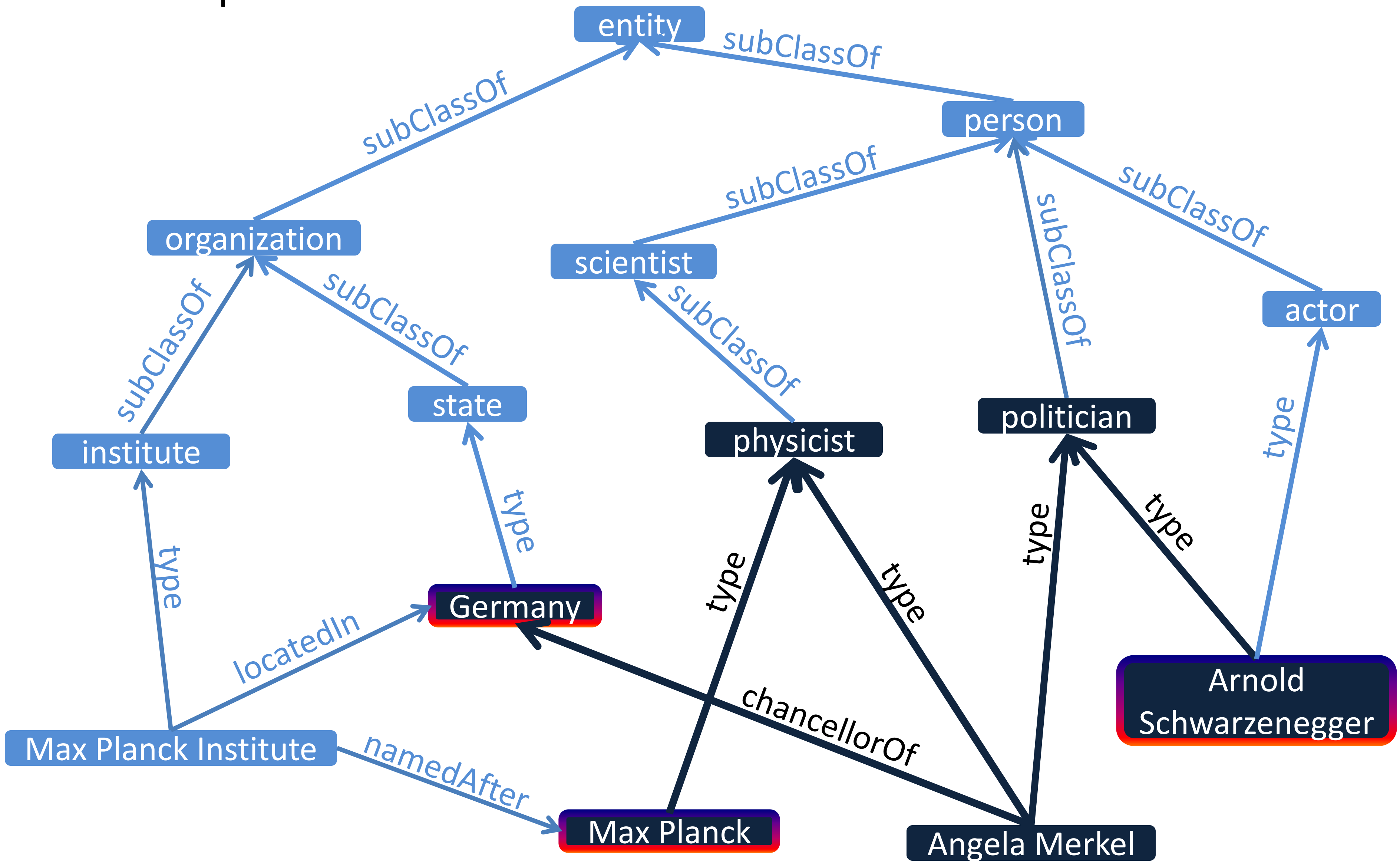
Example: Phase II



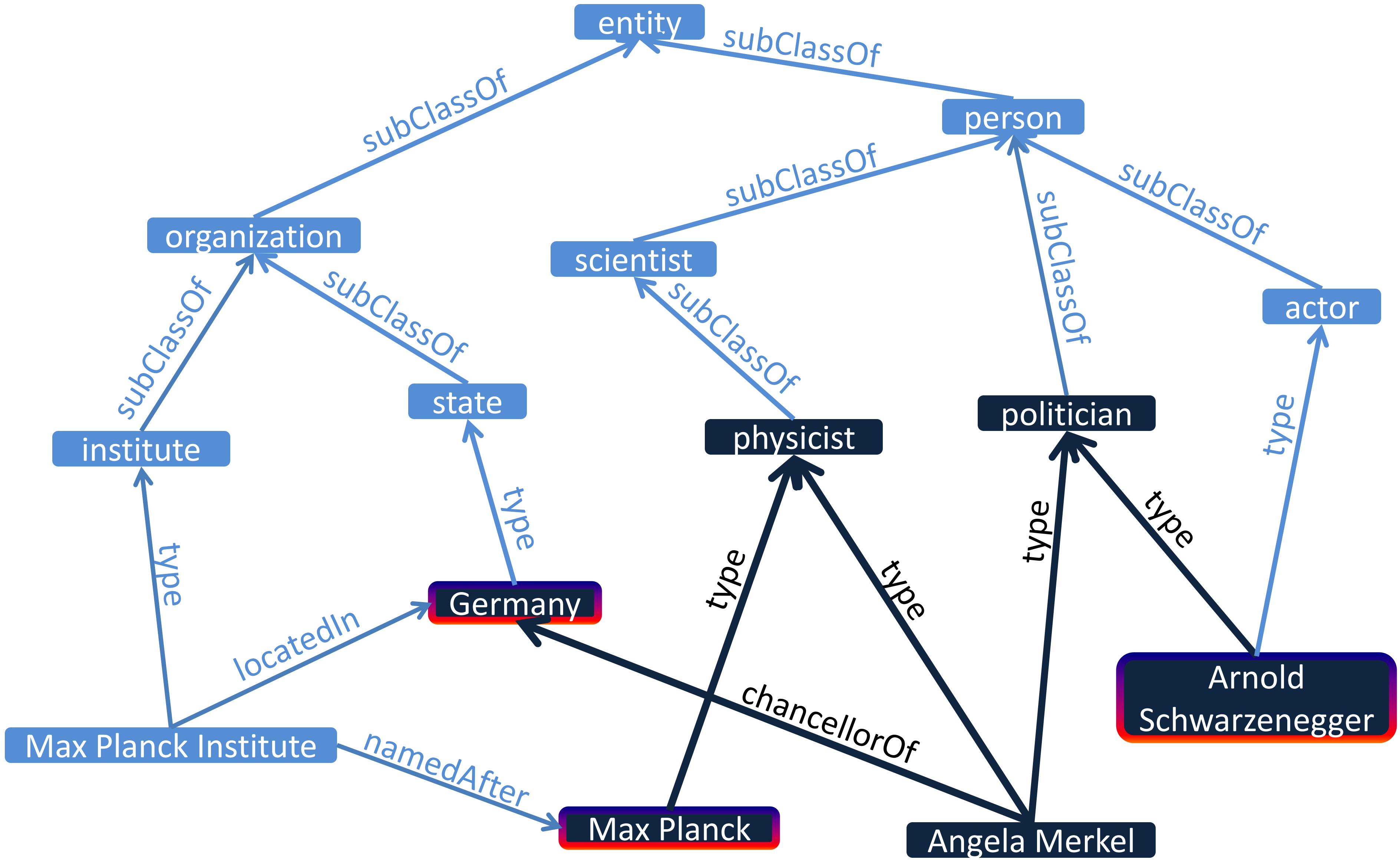
Example: Phase II



Example: Phase II

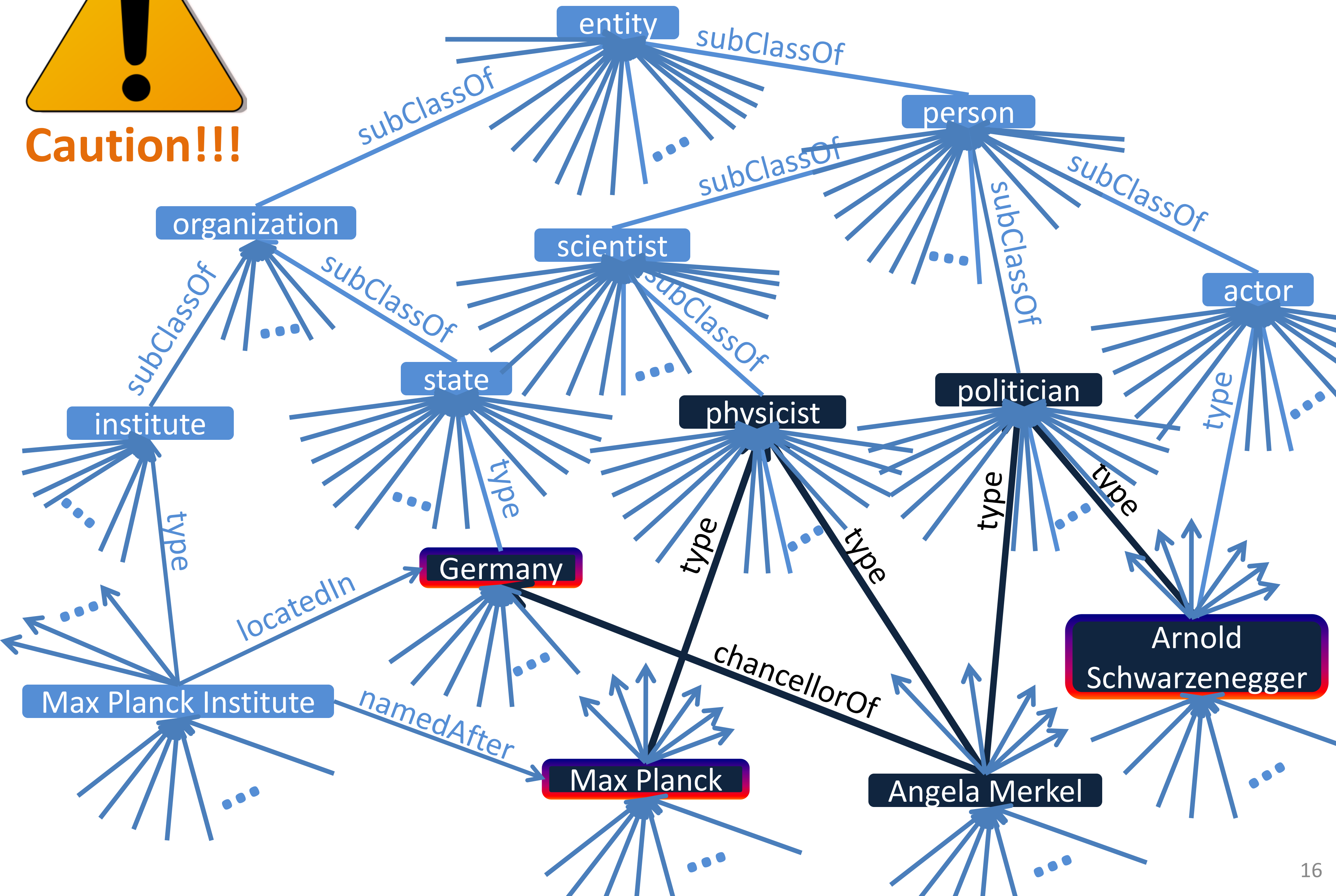


So what? …can't we search for this tree right away?



Caution!!!

Search space should be explored carefully!



STAR: Shortest Path Heuristic

Algorithm 2 *findShortestPath*$(V(T_1), V(T_2), lp)$

for all $v \in V$ **do**
  if $v \in V(T_1)$ **then** $d_1(v) = 0$ **else** $d_1(v) = \infty$
  if $v \in V(T_2)$ **then** $d_2(v) = 0$ **else** $d_2(v) = \infty$
end for
PriorityQueue $Q_1 = V(T_1)$ //ordered by inc. distance d_1
PriorityQueue $Q_2 = V(T_2)$ //ordered by inc. distance d_2
$current$=1
$other$=2
repeat
  if *fringe*(Q_{other})<*fringe*$(Q_{current})$ **then**
    swap$(current, other)$
  end if
  $v = Q_{current}.dequeue()$
  if $d_{current}(v) \geq w(lp)$ **then**
    break
  end if
  for all $(v, v') \in E$ **do**
    if v' *has been dequeued from* $Q_{current}$ **then**
      continue
    end if
    if $d_{current}(v') > d_{current}(v) + w(v, v')$ **then**
      $d_{current}(v') = d_{current}(v) + w(v, v')$
      $v'.predecessor_{current} = v$
    end if
    $Q_{current}.enqueue(v')$
  end for
until $Q_1 = \emptyset \vee Q_2 = \emptyset \vee v \in V(T_{other})$
return *path connecting* T_1 *and* T_2

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

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.

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.

… in theory, and very efficient in practice.

STAR: Top-K Approximate Trees

```
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)

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

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

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

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

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

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