REGULAR PATH QUERIES IN MILLENNIUMDB

Domagoj Vrgoč

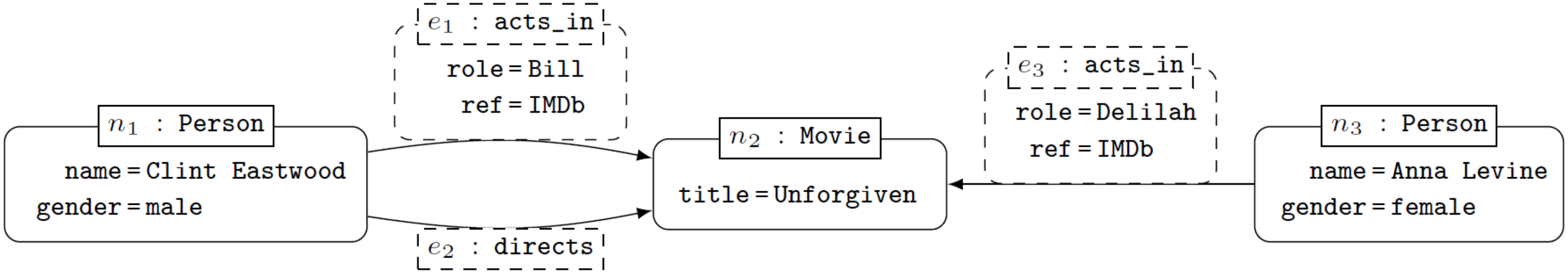
PUC Chile

Institute for Foundational Research on Data

Fundamentos de los datos

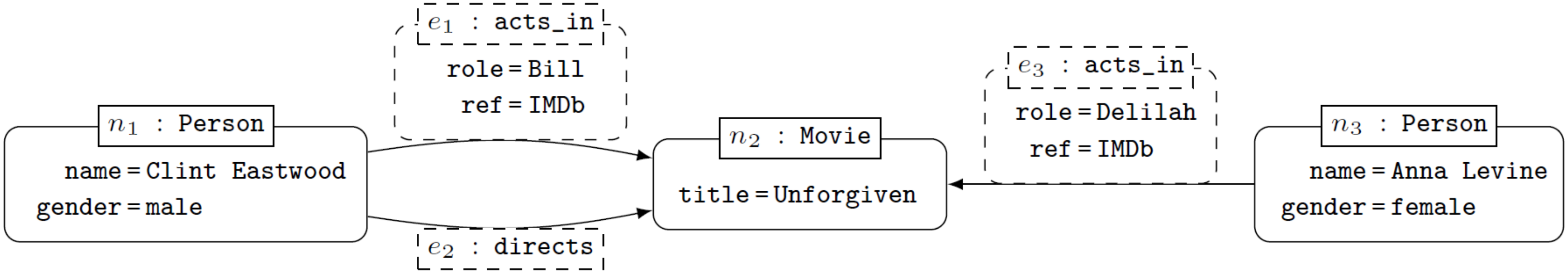
MILLENNIUMDB

- What is MillenniumDB?
  - Open source graph database
  - https://github.com/MillenniumDB/MillenniumDB
  - Based on recent research on wco algorithms and path queries



MILLENNIUMDB – YES, IT IS RELATIONAL

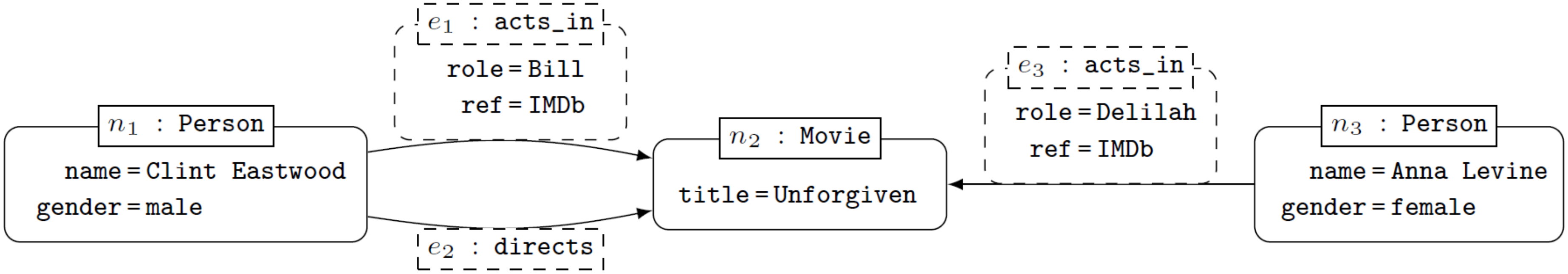
- How is the data stored?



Connections(from,type,to,edgeId)
Properties(object,property,value)
Labels(object,label)

MILLENNIUMDB – YES, IT IS RELATIONAL

- How is the data stored?



Connections(n1,acts_in,n2,e1)
Properties(e1,role,"Bill")
Labels(n1,Person)

MILLENNIUMDB – SOME DETAILS

- Characteristics:
  - Relational storage
  - B+tree indices
  - Pipelined execution
  - Graceful timeouts/query interrupts
  - WCO/Sellinger/Greedy for joins
  - Automata guided search for paths

Is it any good?

BENCHMARKING

- Wikidata Truthy
  - 1.25B edges
  - 92M nodes
  - 46M edge types

- Wikidata query log
  - Let's make it interesting: code 500 queries
  - Around 800 join queries
  - Around 1700 property path queries
  - Timeout set to 5min; single core machine, 128GB RAM

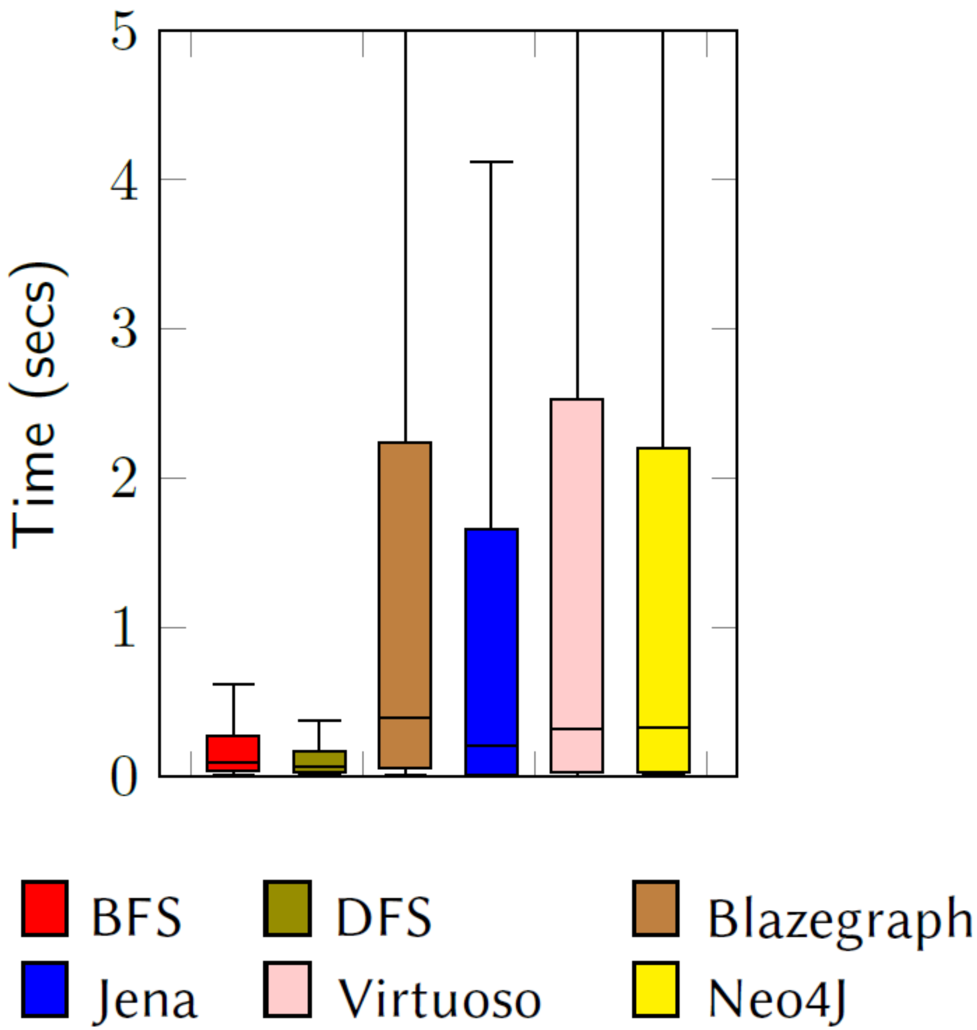
BENCHMARKING

- Wikidata Truthy
  - 1.25B edges
  - 92M nodes
  - 46M edge types

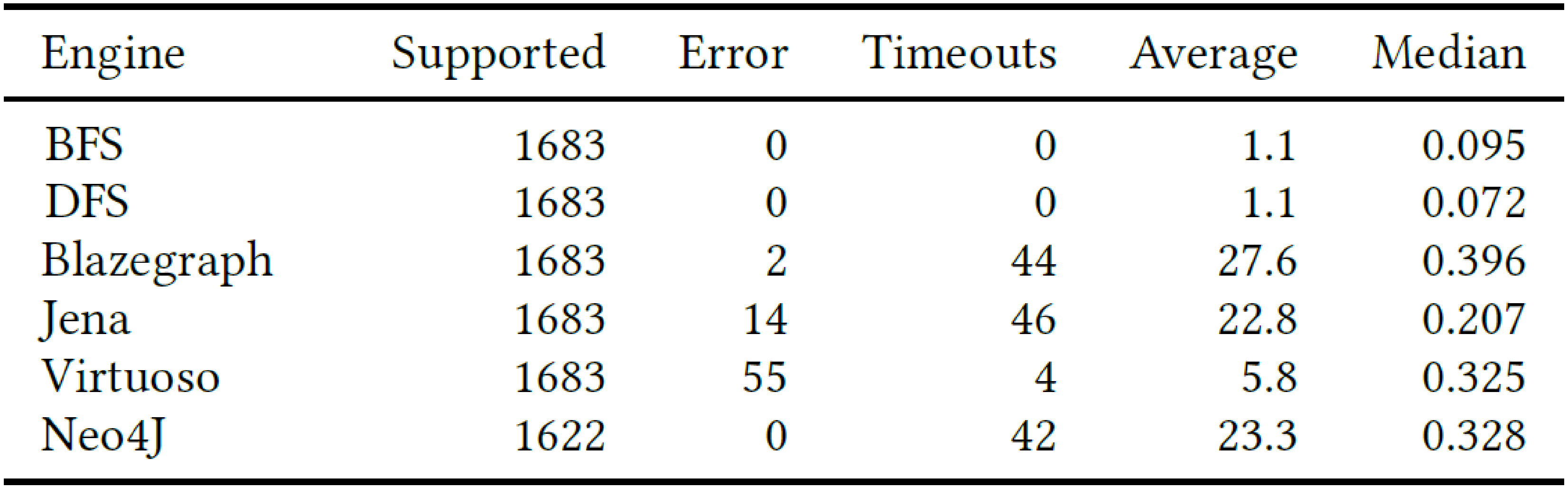
- Wikidata query log
  - Let's make it interesting: code 500 queries
  - Around 800 join queries
  - Around 1700 property path queries
  - Timeout set to 5min; single core machine, 128GB RAM

I will just talk about this

RESULTS

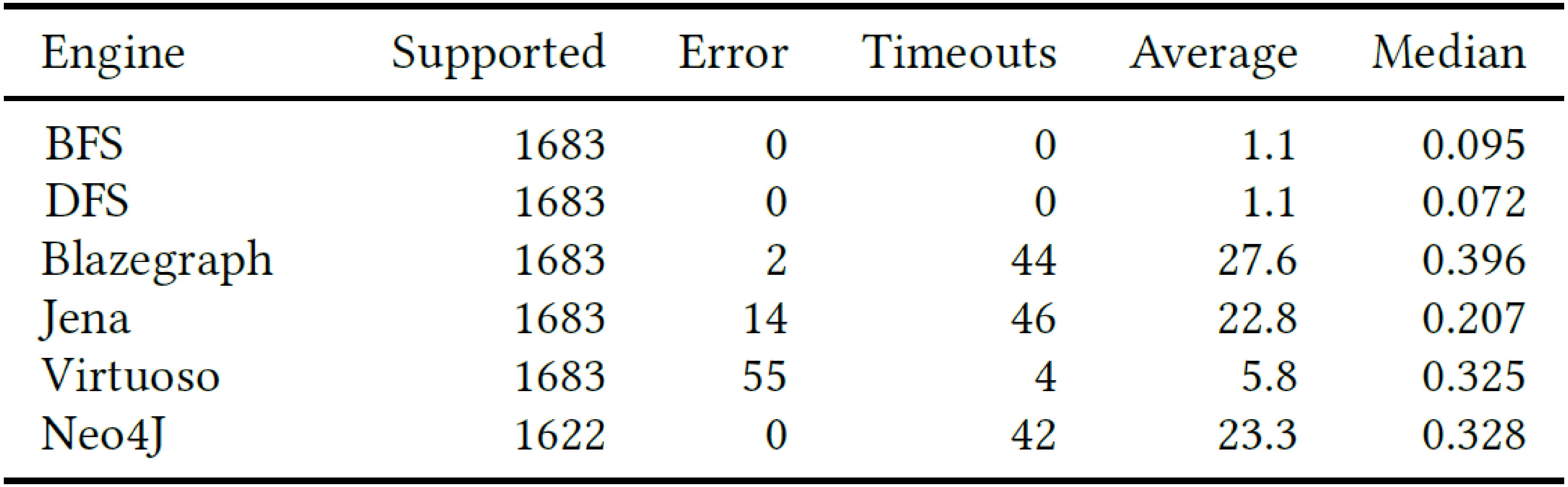


RESULTS



Engine	Supported	Error	Timeouts	Average	Median
BFS	1683	0	0	1.1	0.095
DFS	1683	0	0	1.1	0.072
Blazegraph	1683	2	44	27.6	0.396
Jena	1683	14	46	22.8	0.207
Virtuoso	1683	55	4	5.8	0.325
Neo4J	1622	0	42	23.3	0.328

RESULTS



Engine	Supported	Error	Timeouts	Average	Median
BFS	1683	0	0	1.1	0.095
DFS	1683	0	0	1.1	0.072
Blazegraph	1683	2	44	27.6	0.396
Jena	1683	14	46	22.8	0.207
Virtuoso	1683	55	4	5.8	0.325
Neo4J	1622	0	42	23.3	0.328

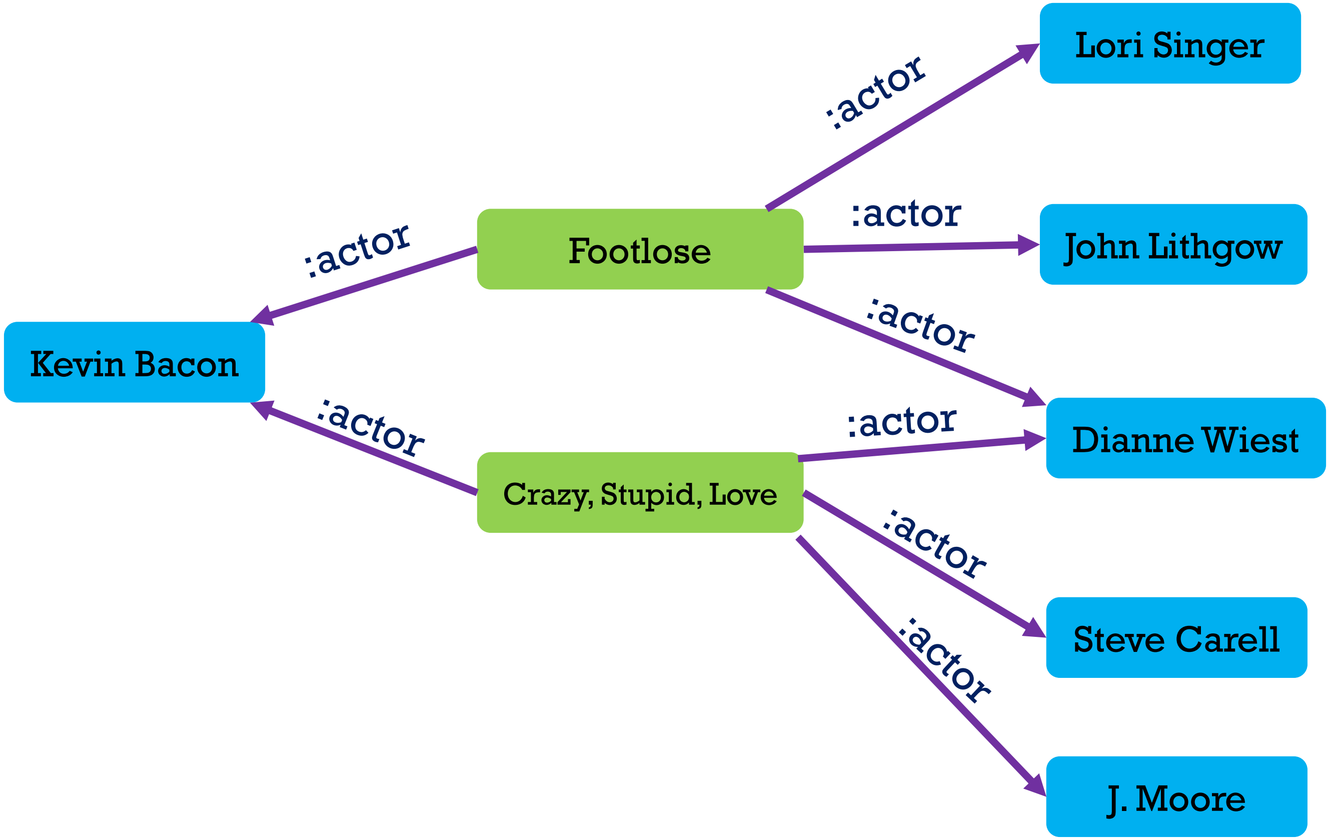
What is going on?

HOW TO EVALUATE PATH QUERIES?

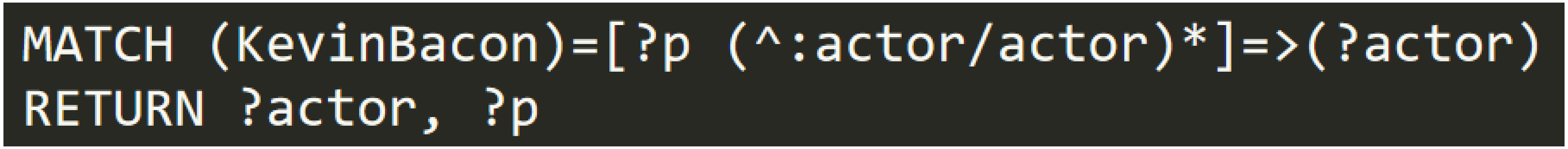
- Theoretician's answer ("This is trivial"): [MW95, CMW87]
  - Graph is an automaton
  - Regular expression is an automaton
  - Do the cross product (on-the-fly to be "efficient")
  - Do reachability check from start states to end states

- Which algorithms can do this?
  - BFS
  - DFS
  - A*
  - IDDFS
  - ...

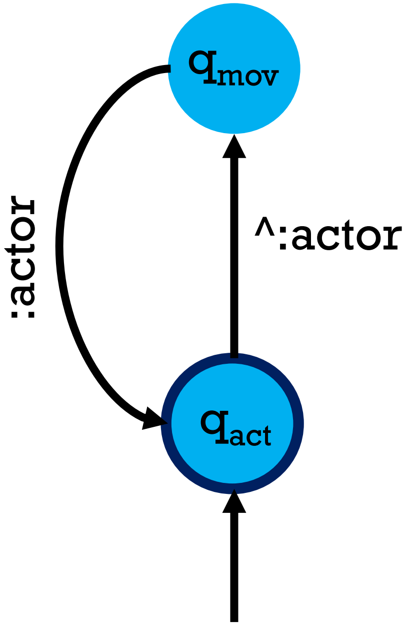
HOW DOES THIS ACTUALLY WORK?



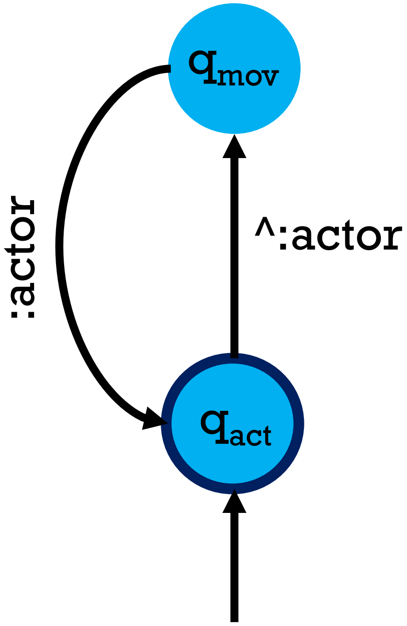
HOW DOES THIS ACTUALLY WORK?



```
MATCH (KevinBacon)=[?p (^:actor/actor)*]=>(?actor)
RETURN ?actor, ?p
```



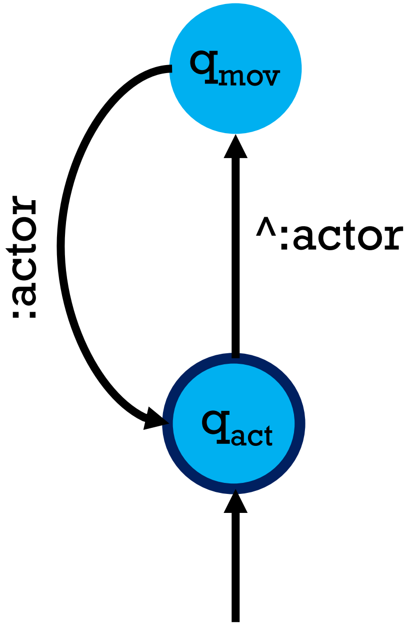
HOW DOES THIS ACTUALLY WORK - BFS



q_{act}

Kevin Bacon

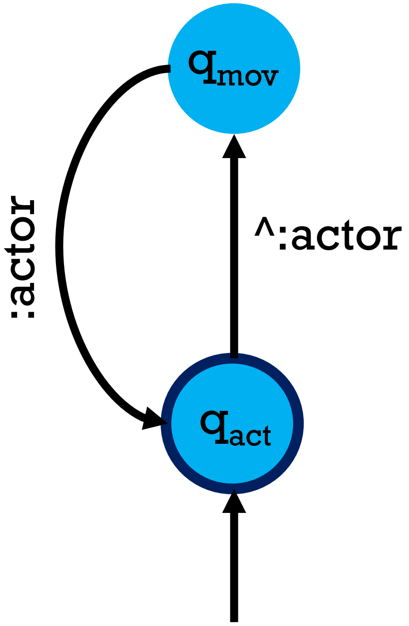
HOW DOES THIS ACTUALLY WORK - BFS

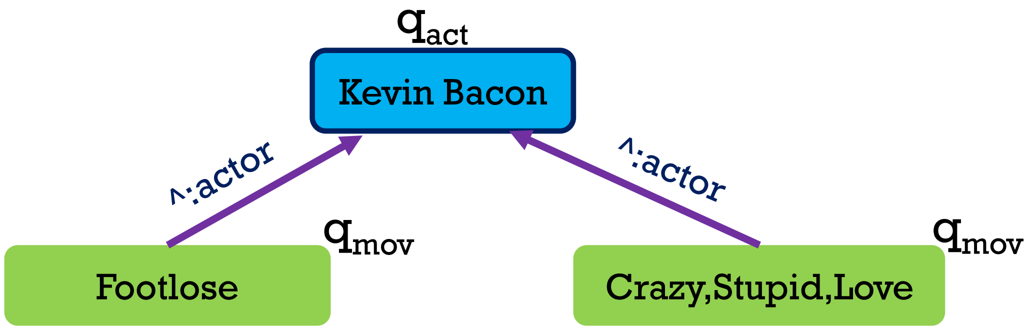


q_{act}

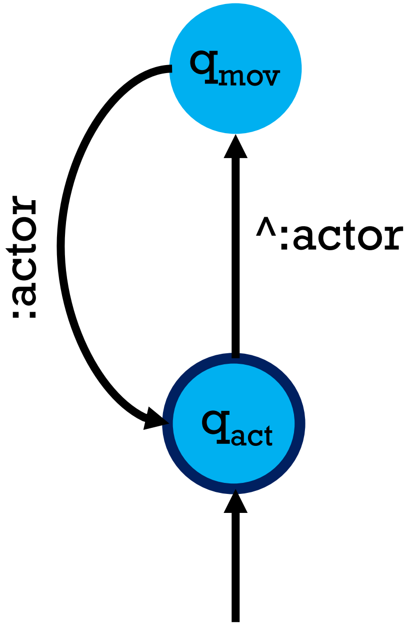
Kevin Bacon

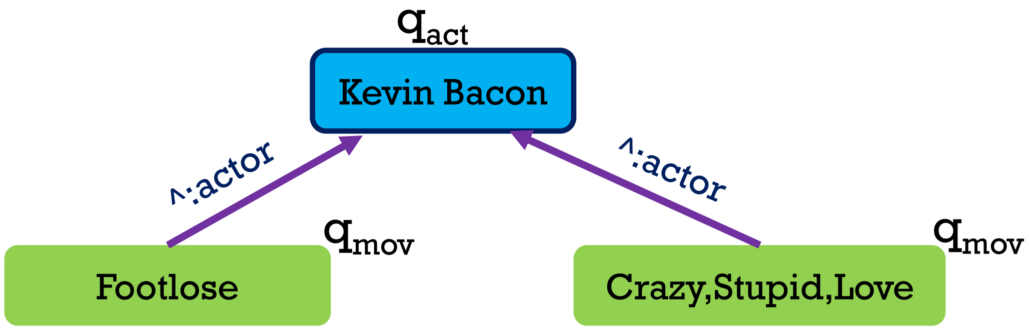
HOW DOES THIS ACTUALLY WORK - BFS





HOW DOES THIS ACTUALLY WORK - BFS

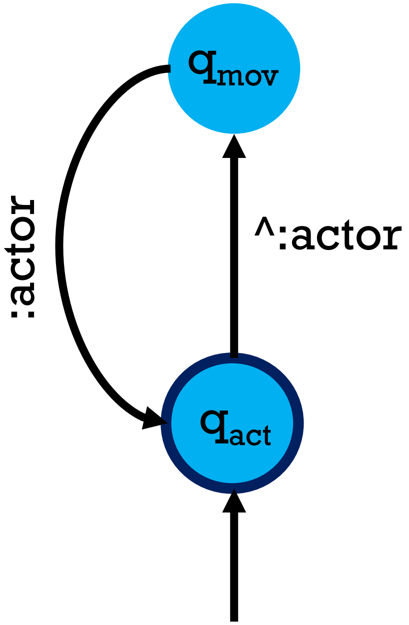


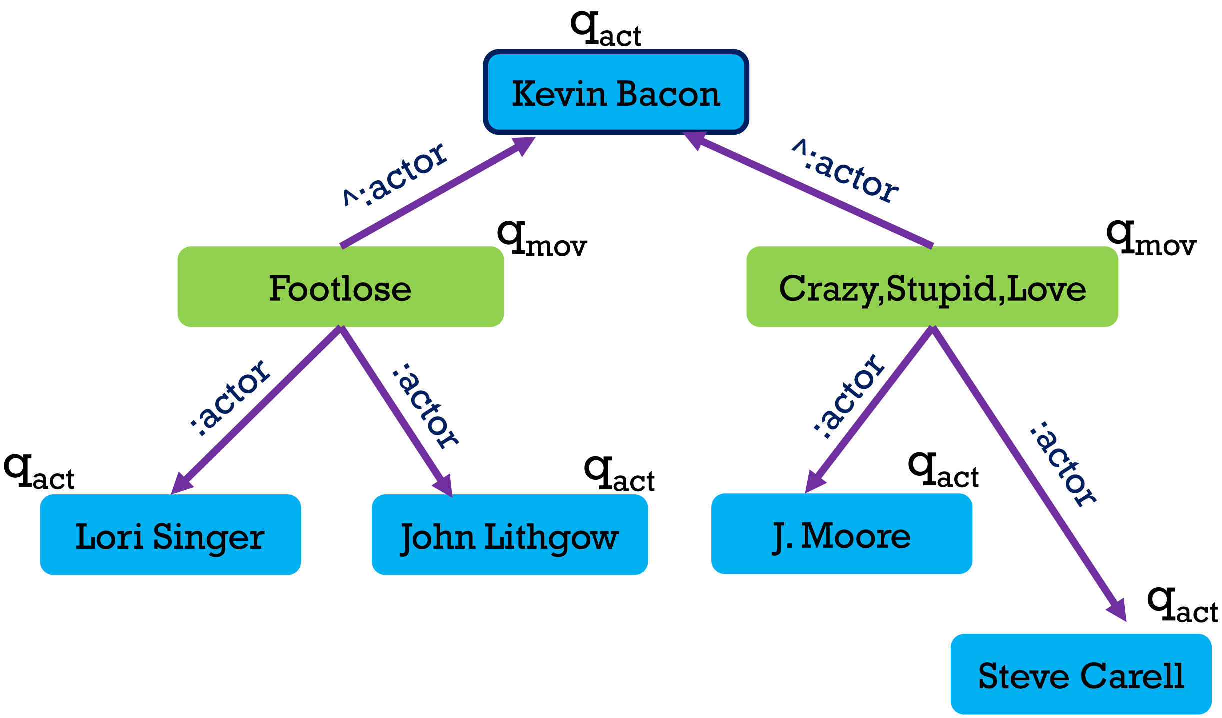


This is just B+tree search; **Connection**$^{-}$**(KevinBacon,actor,source,eId)**

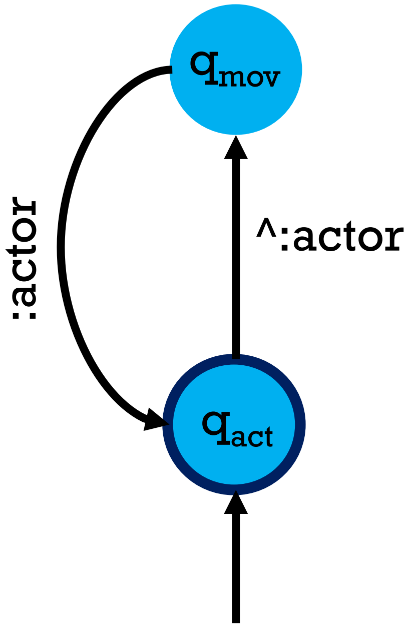
Requires single page pinned in the buffer (for BFS)!

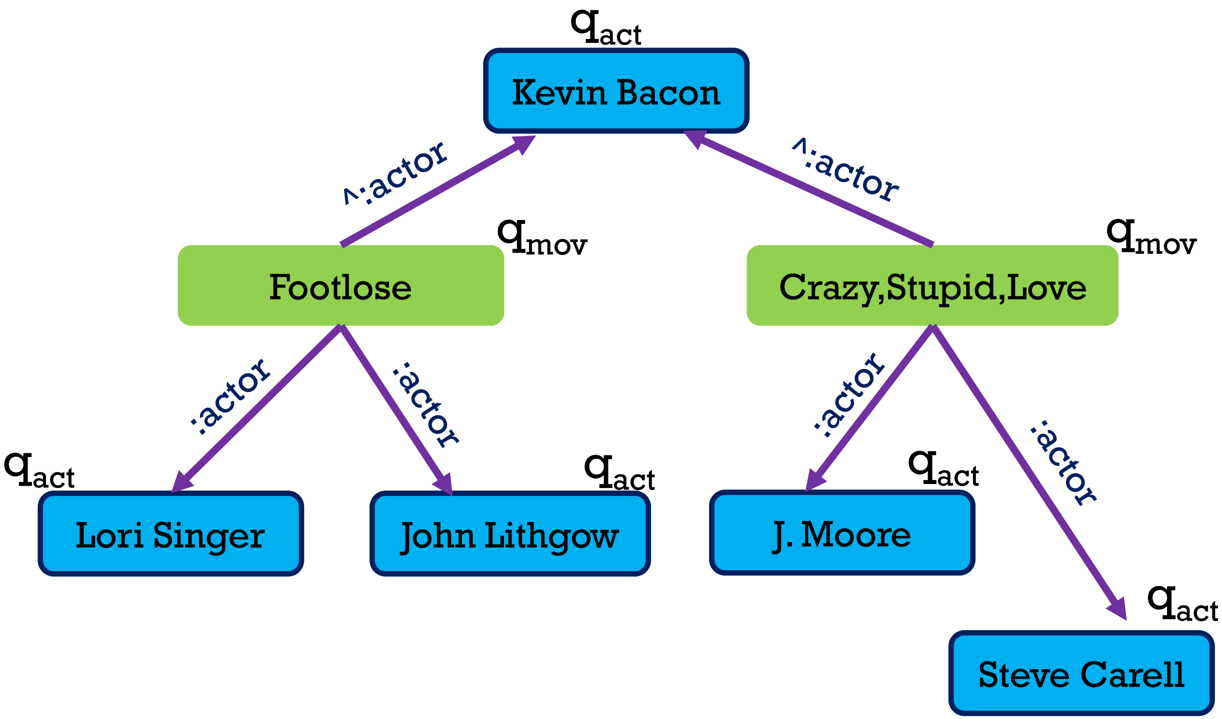
HOW DOES THIS ACTUALLY WORK - BFS



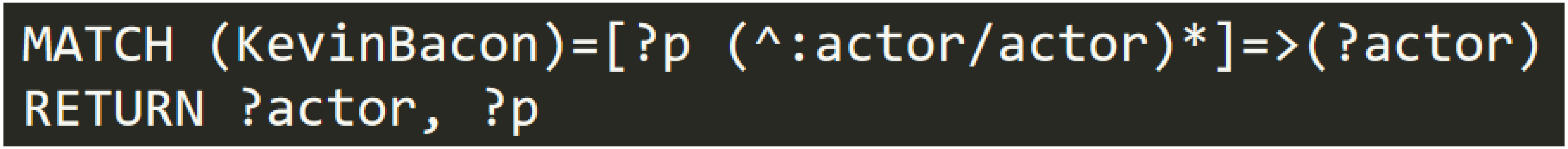


HOW DOES THIS ACTUALLY WORK - BFS





WHAT WILL WE RETURN TO THE USER?



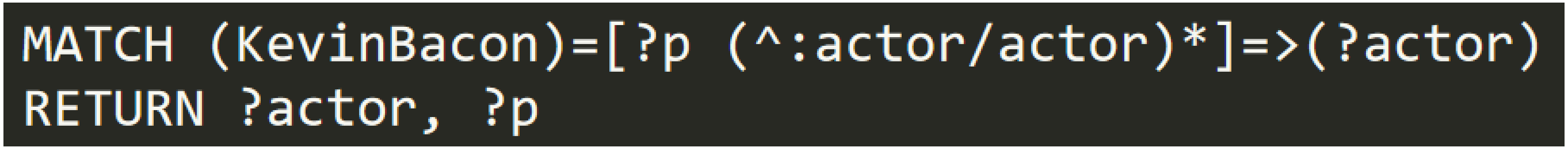
```
MATCH (KevinBacon)=[?p (^:actor/actor)*]=>(?actor)
RETURN ?actor, ?p
```

Option 1: Just the enpoint pairs (x,y)

Option 2: Enpoint pairs plus a single path/witness

Option 3: For each endpoint pair all shortest paths connecting them

WHAT WILL WE RETURN TO THE USER?



```
MATCH (KevinBacon)=[?p (^:actor/actor)*]=>(?actor)
RETURN ?actor, ?p
```

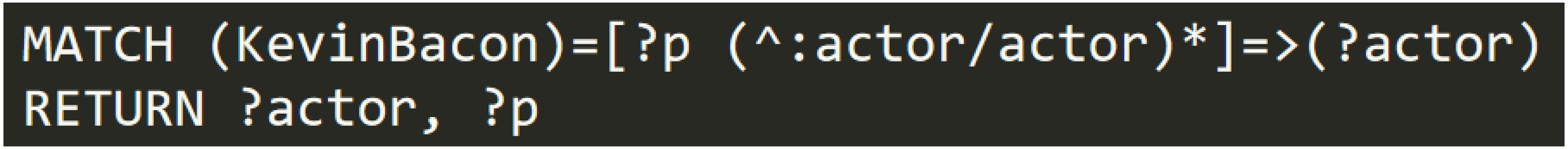
Option 1: Just the enpoint pairs (x,y)

Option 2: Enpoint pairs plus a single path/witness

Option 3: For each endpoint pair all shortest paths connecting them

What type of a path (walk, trail, simple)?

WHAT WILL WE RETURN TO THE USER?



```
MATCH (KevinBacon)=[?p (^:actor/actor)*]=>(?actor)
RETURN ?actor, ?p
```

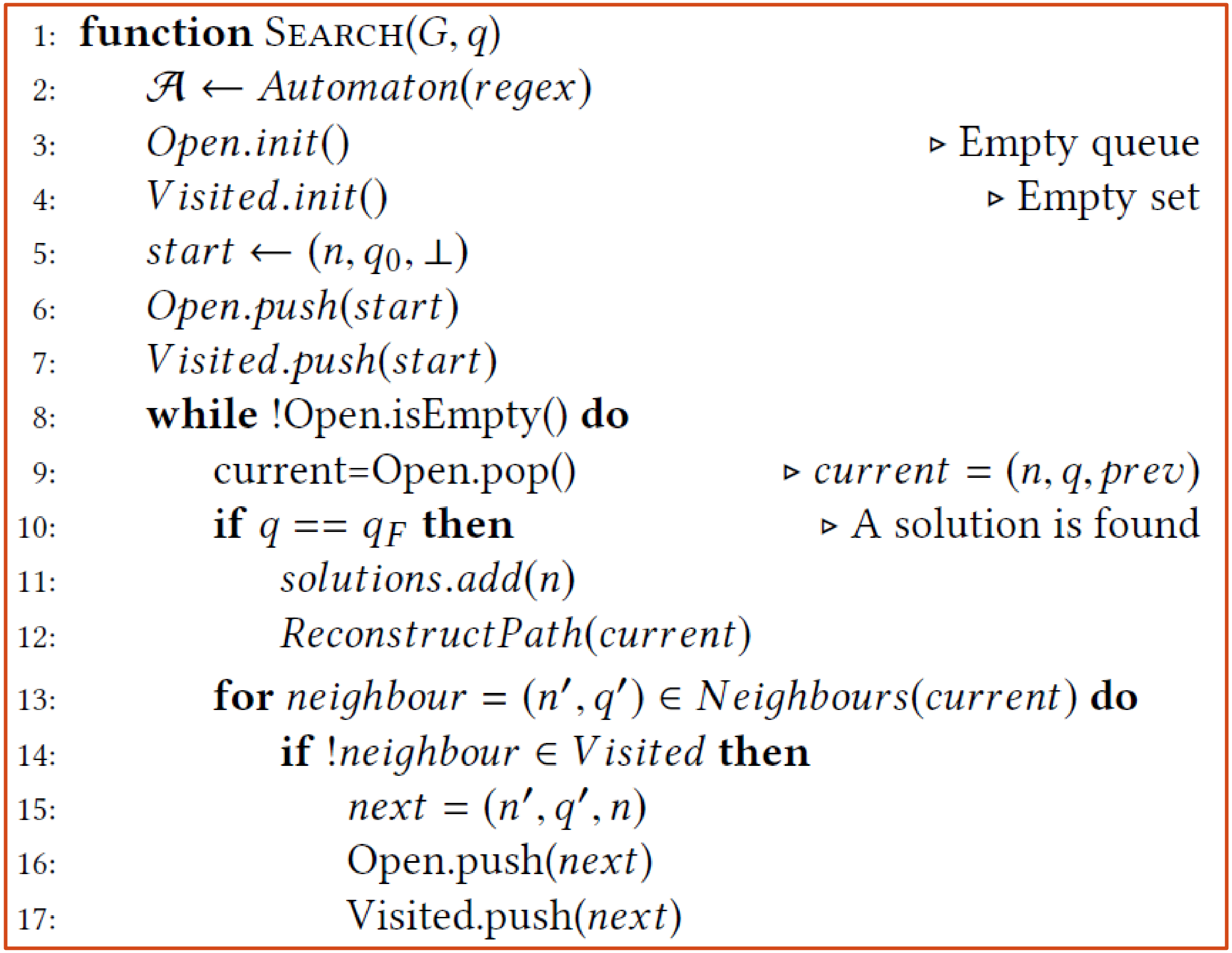
Option 1: Just the enpoint pairs (x,y)

Option 2: Enpoint pairs plus a single path/witness

Option 3: For each endpoint pair all shortest paths connecting them

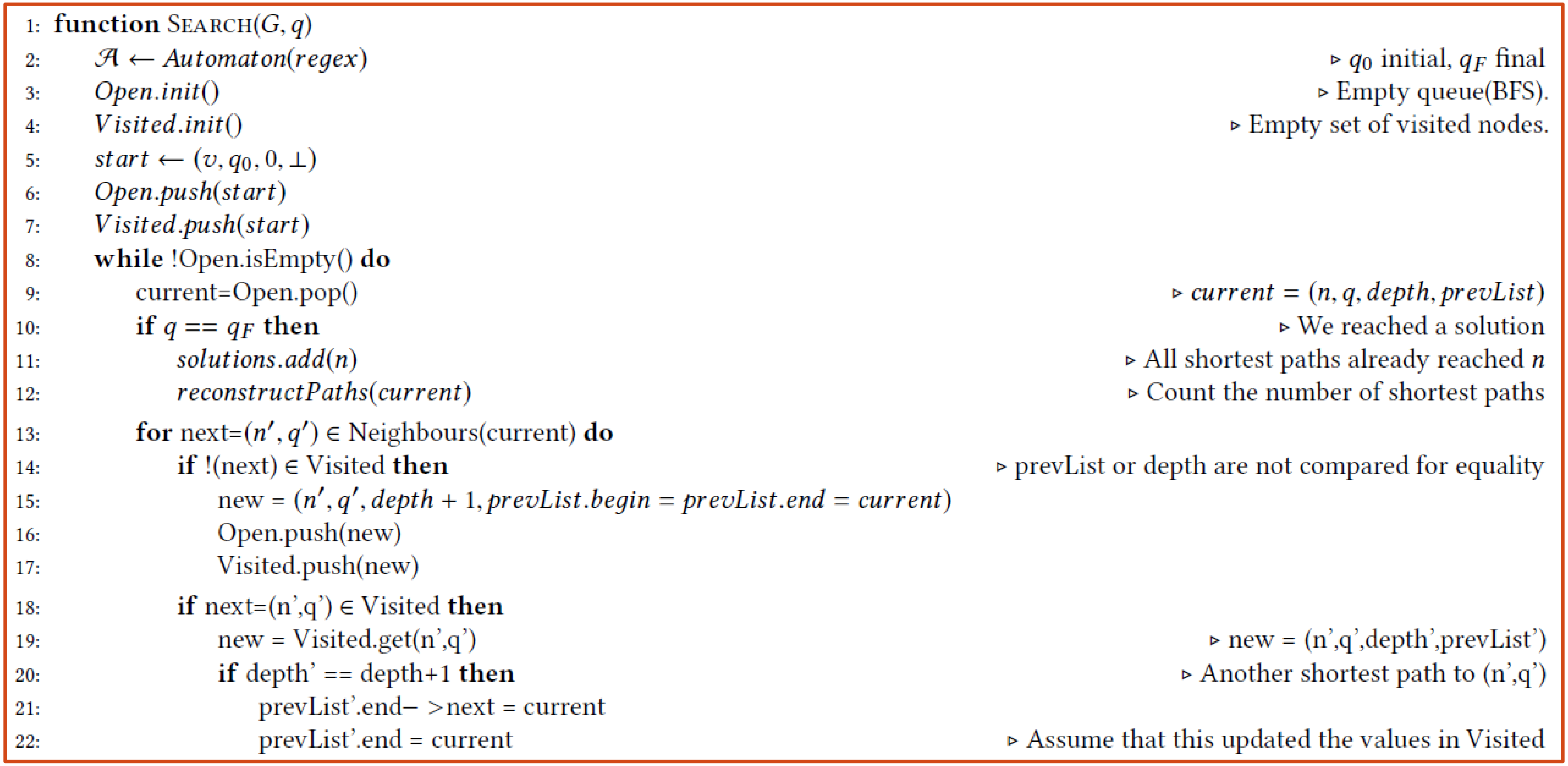
I will look at walks (any path)!

BFS – ALSO A PATH (ONE PER PAIR)



```
function SEARCH($G, q$)
    $\mathcal{A} \leftarrow Automaton(regex)$
    $Open.init()$                                  ▷ Empty queue
    $Visited.init()$                               ▷ Empty set
    $start \leftarrow (n, q_0, \perp)$
    $Open.push(start)$
    $Visited.push(start)$
    while !Open.isEmpty() do
        current=Open.pop()                         ▷ $current = (n, q, prev)$
        if $q == q_F$ then                         ▷ A solution is found
            $solutions.add(n)$
            $ReconstructPath(current)$
        for $neighbour = (n', q') \in Neighbours(current)$ do
            if !$neighbour \in Visited$ then
                $next = (n', q', n)$
                Open.push($next$)
                Visited.push($next$)
```

BFS – ALL SHORTEST PATHS



```
function SEARCH($G, q$)
    $\mathcal{A} \leftarrow Automaton(regex)$                                ▷ $q_0$ initial, $q_F$ final
    $Open.init()$                                                          ▷ Empty queue(BFS).
    $Visited.init()$                                          ▷ Empty set of visited nodes.
    $start \leftarrow (v, q_0, 0, \perp)$
    $Open.push(start)$
    $Visited.push(start)$
    while !Open.isEmpty() do
        current=Open.pop()                                     ▷ $current = (n, q, depth, prevList)$
        if $q == q_F$ then                                                   ▷ We reached a solution
            $solutions.add(n)$                                  ▷ All shortest paths already reached $n$
            $reconstructPaths(current)$                            ▷ Count the number of shortest paths
        for next=$(n', q')$ ∈ Neighbours(current) do
            if !(next) ∈ Visited then                   ▷ prevList or depth are not compared for equality
                new = $(n', q', depth + 1, prevList.begin = prevList.end = current)$
                Open.push(new)
                Visited.push(new)
            if next=(n',q') ∈ Visited then
                new = Visited.get(n',q')                                    ▷ new = (n',q',depth',prevList')
                if depth' == depth+1 then                                ▷ Another shortest path to (n',q')
                    prevList'.end– >next = current
                    prevList'.end = current                   ▷ Assume that this updated the values in Visited
```

A FEW COMMENTS ON PATHS

How do we return paths?

- Basicaly a list of node/edge pairs
- Internally this is the structure Wim spoke about

What else could be done?

- Parallel execution
- Start in the middle approach
- Trails, simple paths
- Data comparisons (already done really)

MORE DETAILS

MillenniumDB source code:

- https://github.com/MillenniumDB/MillenniumDB

Explanation of the algorithms:

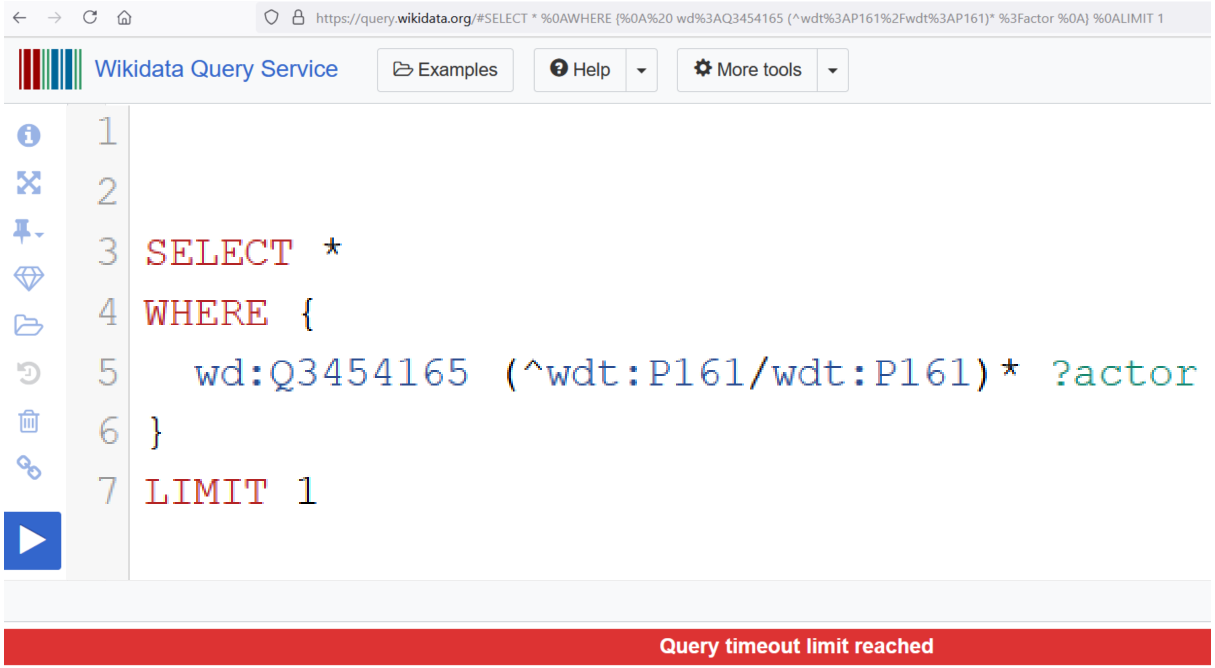
- https://arxiv.org/abs/2204.11137

Benchmarks:

- https://github.com/MillenniumDB/benchmark
- https://github.com/MillenniumDB/WDBench

IS THIS HOW PATHS ARE IMPLEMENTED?

- SPARQL
  - Endpoints/set semantics
  - No counting paths(standard)



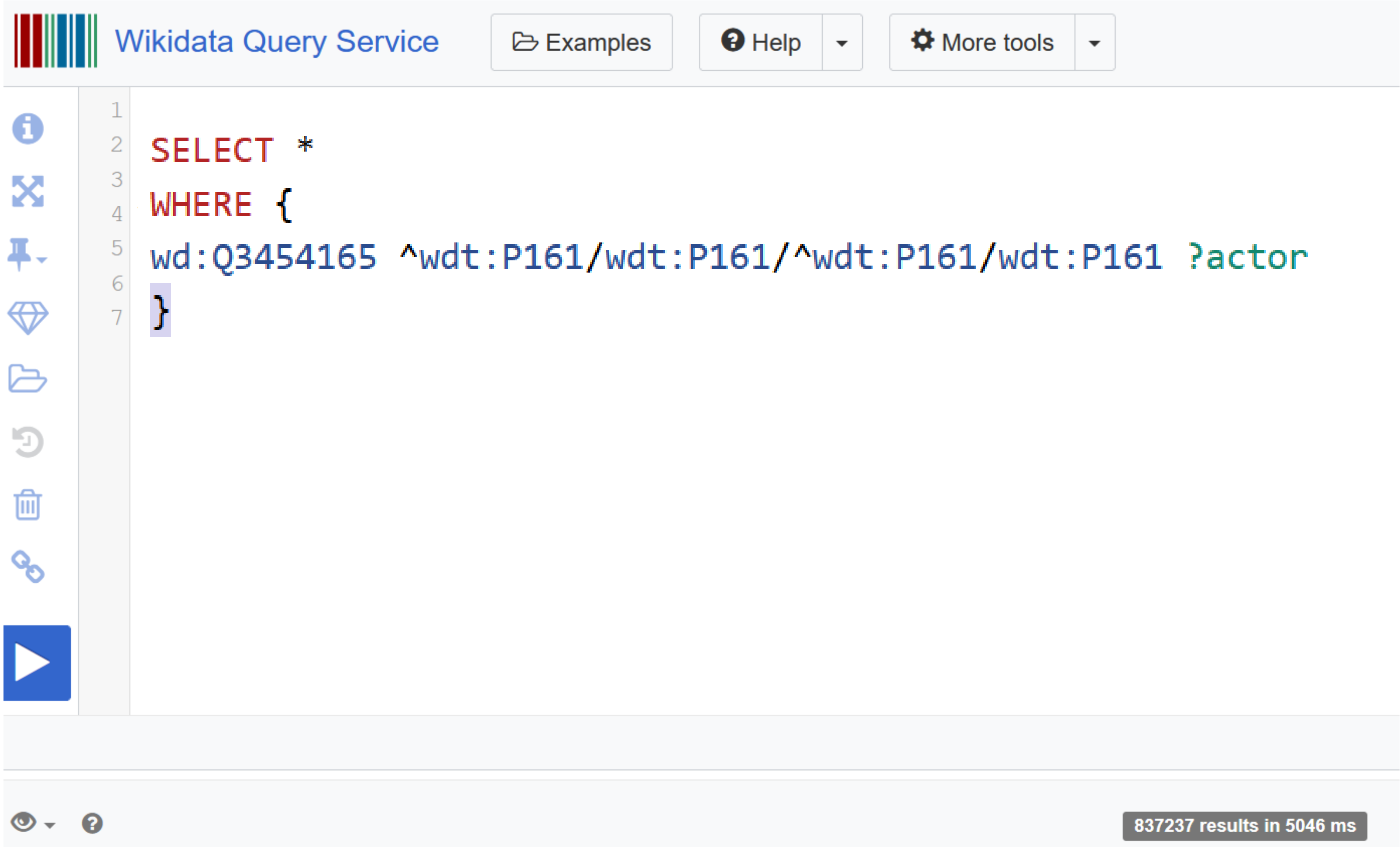
https://query.wikidata.org/#SELECT * %0AWHERE {%0A%20 wd%3AQ3454165 (^wdt%3AP161%2Fwdt%3AP161)* %3Factor %0A} %0ALIMIT 1

```


SELECT *
WHERE {
  wd:Q3454165 (^wdt:P161/wdt:P161)* ?actor
}
LIMIT 1
```

Query timeout limit reached

SPARQL'S ODDITIES



```

SELECT *

WHERE {

wd:Q3454165 ^wdt:P161/wdt:P161/^wdt:P161/wdt:P161 ?actor

}
```

837237 results in 5046 ms

THE MESSAGE

Good baselines are really really really really important!!!

Thank you!