How Graphs and Java make graphhopper efficient and fast

By Peter @timetabling

Berlin Buzzwords, 2014-05-27

|_

Available at graphhopper.com/public/slides

How int[][] helped GraphHopper scaling

~~How Graphs and Java make GraphHopper efficient and fast~~

By Peter @timetabling

Berlin Buzzwords, 2014-05-27

Available at graphhopper.com/public/slides

Components of an Online Map

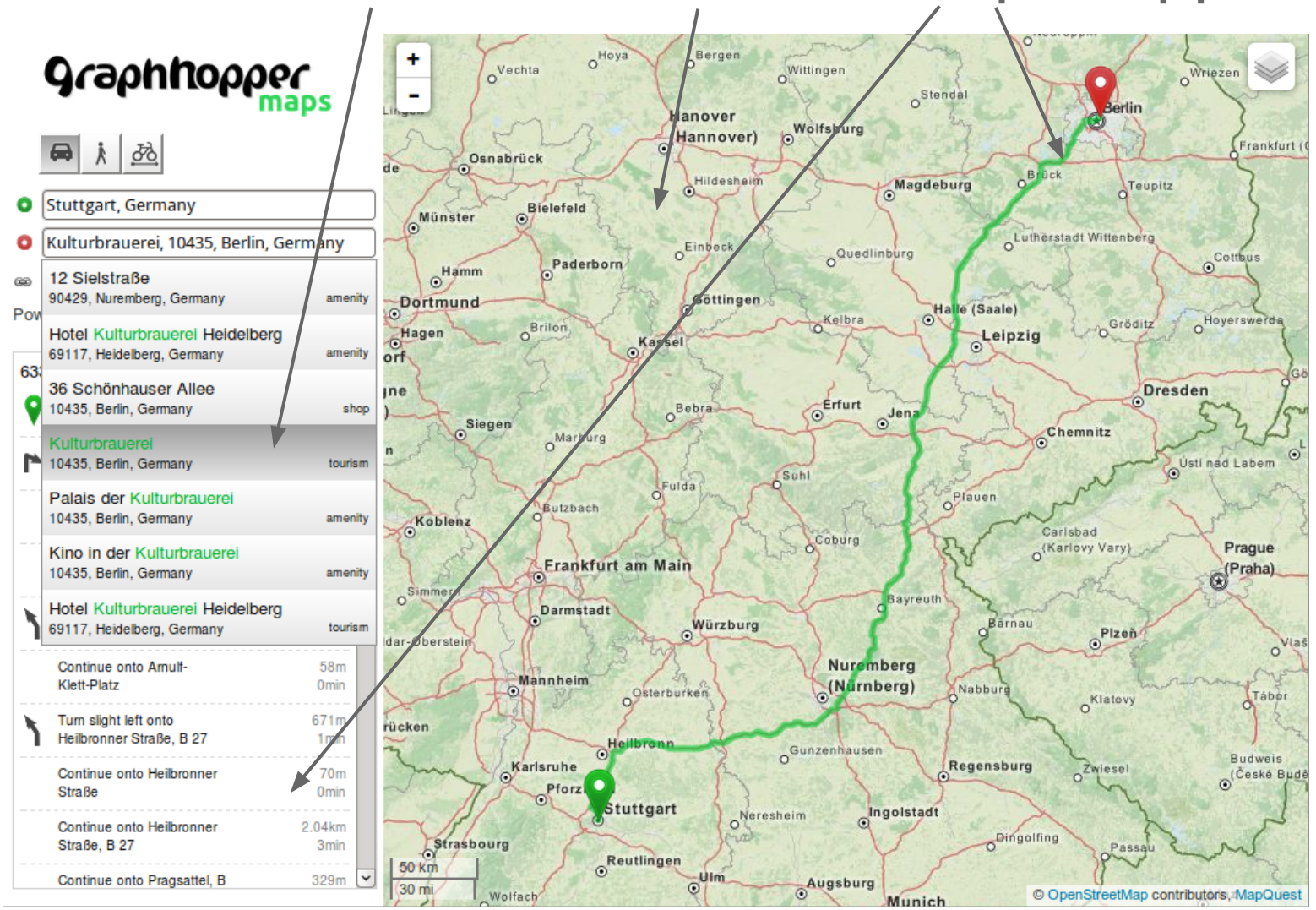
A full “maps” application requires:

1. Drawing: Display map from vector or raster data
2. Geocoding: Search address, get GPS coordinates
   E.g. we use photon powered by ElasticSearch

3. **Routing**: find best paths between coordinates
   → GraphHopper is all about routing!

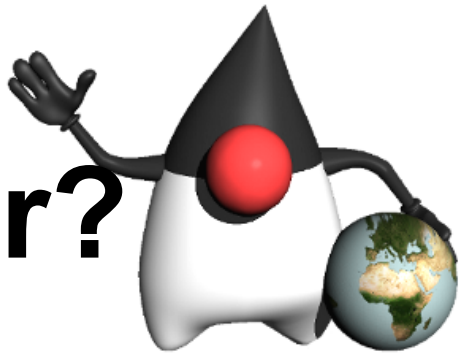
GraphHopper Maps

= Address Search* + Tiles + GraphHopper



graphhopper.com/maps

What is GraphHopper?



1. Open Source & fast road routing library and server
2. Written in Java: runs on Server, Desktop, Android, …
   new: offline in the Browser, Raspberry Pi and iOS
3. Very memory-efficient but still has an easy to use API
4. The Low-level API is built to be flexible
5. Handles OpenStreetMap data by default
6. Business-friendly: Apache License and we offer Consulting & Support
7. Many unit, integration and load tests

What is GraphHopper?

⇨ **Hackable & Flexible!**

You can try different implementations for algorithms, use case (social graphs), storage, ...

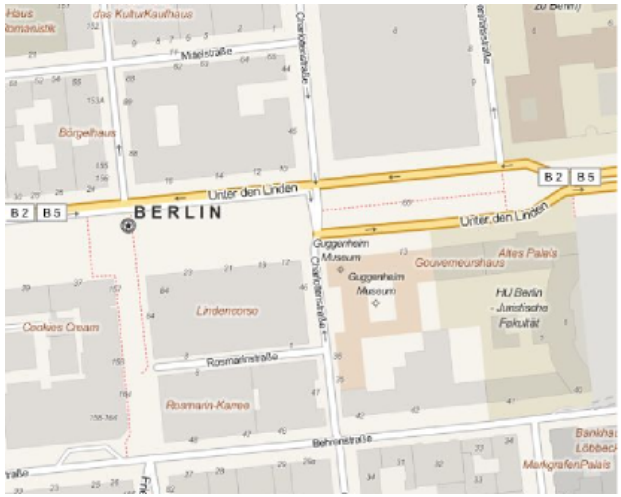
What you can do?

- Point to point routing
- Distance matrix e.g. for logistics
- Outdoor routing for biking/hiking
- Track vehicles via map matching (not included)
- Simulation / Urban planning
- Games or VR (think ‘Scotland Yard’)
- Crisis management
- Graph traversal and statistics

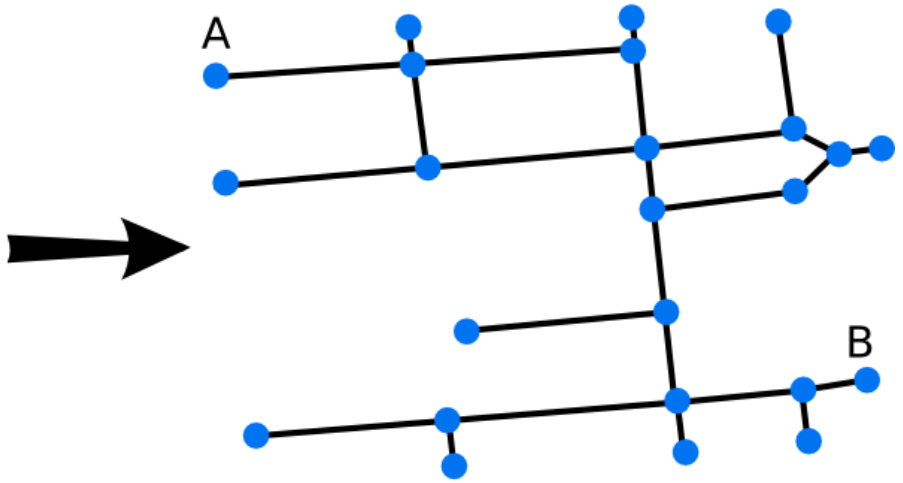
Road Graph

- In a graph we have nodes and edges
- In real world we have junctions and streets
- Edges and nodes have properties like coordinates

Real word network



Road Graph



Why Java?

Normally I answer with:

- Why not?
- I’m stupid and lazy!
- In PHP too many people would have contributed

Why Java?

Today you'll learn the truth:

It is all about tooling!
But also: stupidity!

- C++ compiling is soo slow!
  - yes, javac is faster even through maven ;) !
- Java is easy (for me) to run, test, deploy, debug, profile
- Tried 2 weeks to set up a similar easy tooling in C++/D
- Open Source IDEs for C++ less powerful than Java (read: I'm lazy)
- D is an excellent language but tooling wasn't that good (2012)
- I gave up

Java is slow?

“Knock, knock.”
“Who’s there?”

very long pause…

“Java.”

Java is slow?

compared to what?

GraphHopper finds the best route through entire Europe in under 50ms.

For distance matrix calculations this is <5ms.

Demo!

Java is a memory hog!

compared to *C/C++*

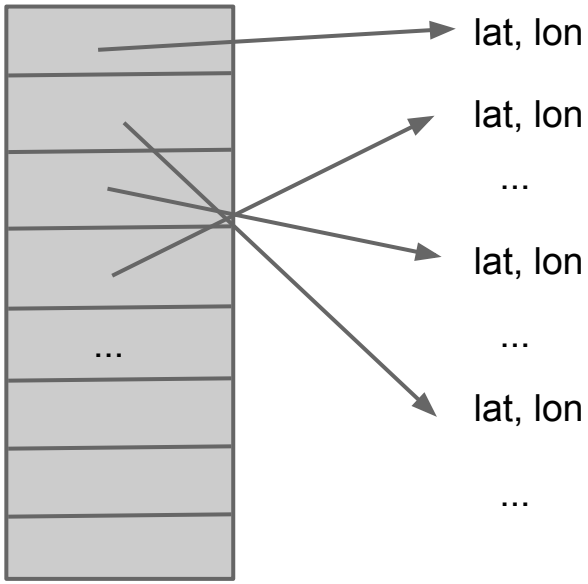
Main reason: no structs in Java!

Oh!

Struct?

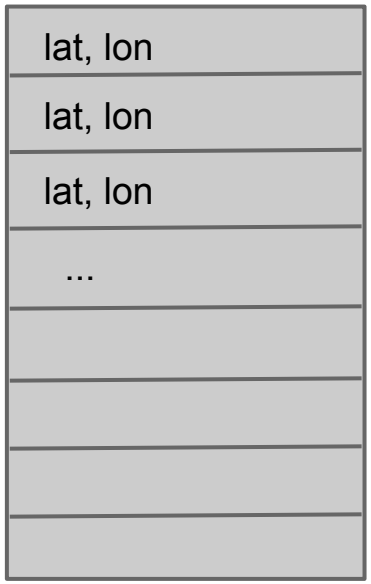
Java array with refs

- additional ref
- cache unfriendly



C++ array with structs

- copy semnatics e.g. if sharing one point in two arrays



- Not that easy to introduce copy semantics in Java
- In Java 9: ValueTypes? Read more about this from John Rose

Until then ...

… we do 2 things to avoid wasting memory

1. **Scale via int[][]**
2. **Flyweight pattern**

1. Scale via int[][]

A simple in-memory key-value storage can be implemented via HashMap<String, Object> in Java

Problems:

- Huge waste of memory due to storing the key
- You need the Object reference (waste especially for small objects)
- Resizing triggers rehashing and costly re-allocation
- Still limited to 2 billion objects

Ideas:

1. Use List<Object> avoids storing the key and the rehashing
2. Use byte[] and (de-)serialization to avoid the Object references
3. Use array of byte[] to append instead of costly costly re-allocation for resizing. But also to allow >2 billion

1. Scale via int[][]

interface DataAccess

Solves:

- less complex access compared to using the raw byte[]
- no 2 billion limit due to ‘long’ key
- can have multiple implementations like byte[][] or **int[][]** (often int[][] is fastest for us)
- can be implemented via array of ByteBuffer => off-heap

  ⟶ very useful for offline navigation on mobile devices (mmap)

Still Problems:

- more complex to access compared to HashMap

How You can scale

- Array-alike access of DataAccess is very specific
- Plenty of more generic solutions for You:
  - MapDB
    provides convenient access via Map interface
  - fasttuple
  - shared-memory-cache
  - larray
  - Java-Lang
- Nearly all (NO-SQL) databases written in Java make use of a similar technique: lucene, hbase, cassandra, ...

2. Flyweight pattern

We use flyweight pattern to traverse the graph
→ avoids creation of new objects due to deserialization

So, instead of:

```
for(RoadEdge edge : graph.getEdges(someNode)) {
    double dist = edge.getDistance();
}
```

… we do:

```
EdgeExplorer explorer = graph.createExplorer();
EdgeIterator iter = explorer.setBaseNode(someNode);
while(iter.next()) {
    double dist = iter.getDistance();
}
```

Why creating a specialized Graph DB?

- neo4j?
- orientdb?
- lucene? (Lumeo)

No, because:

- We needed a very fast and only specialized graph storage!
- Has to run on mobile devices
- Wasn’t fun but necessary

Do your own benchmarks

- Don’t believe me or random benchmarks in the www
- Do your own benchmarks
- But do it correctly! Aleksey Shipilëv, 2009, in response to my microbenchmarking post:

  “*The technique described in this post is ultimately broken. It also contradicts with the best practices of measuring the Java performance.”*

He referred in one of his talks to my post as pitfall #3.
Ouch! Avoid “learning by shame & pain” and try:

- JMH harness for microbenchmarks
- jcstress concurrency stress tests
- Profilers like Yourkit/NetBeans/...

Dijkstra

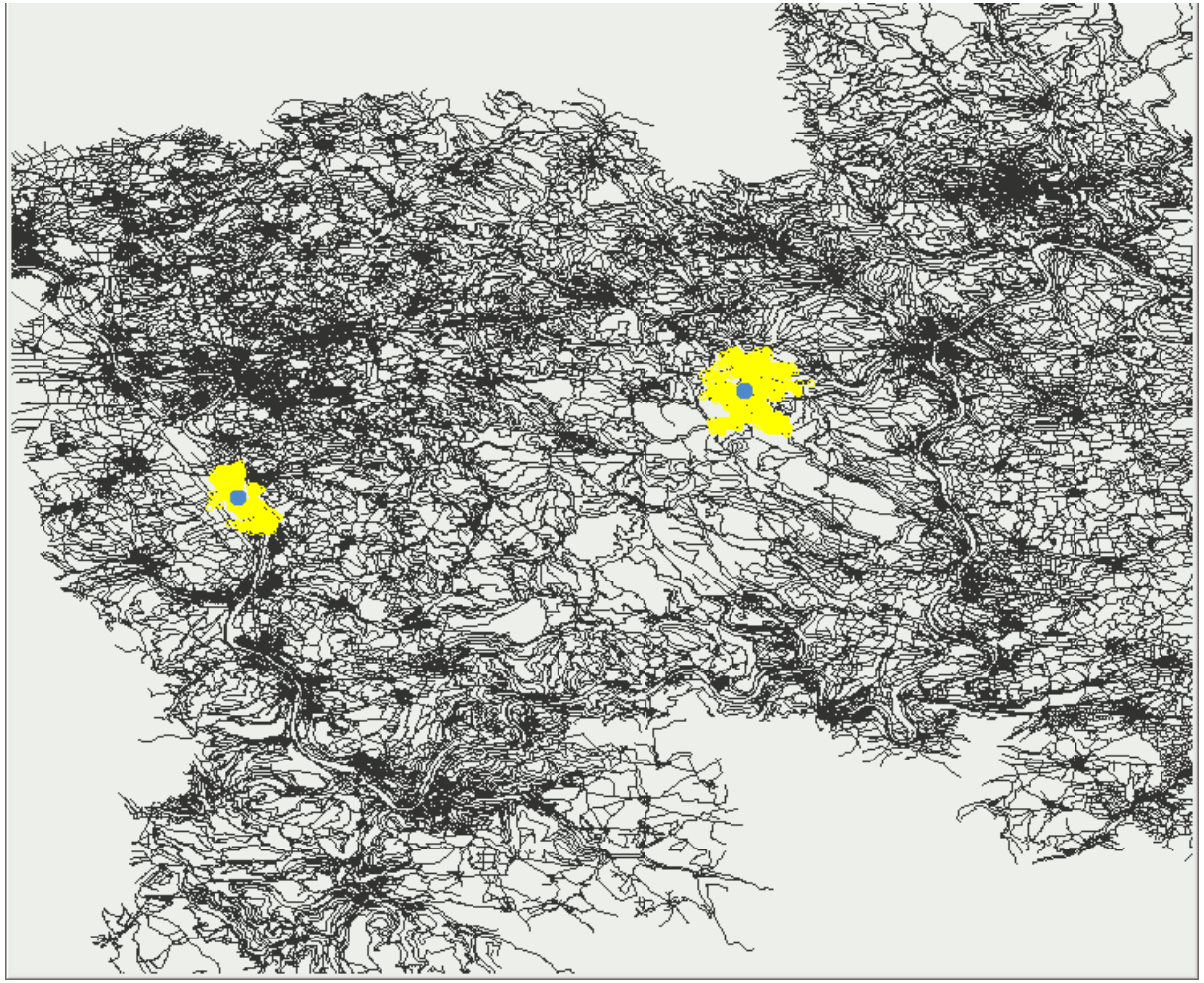
→ **Input**: one start and one end node

1. nodeX := start node
2. Get all neighboring nodes of nodeX
3. Put distance of edges for those nodes into a priority queue
4. later steps: add old distance
5. nodeX : getMin(priority queue)
6. Go to 1, break if nodeX == end node

→ **Output**: Smallest distance from start to end

Get final path via shortest path tree

Bidirectional Dijkstra



Contraction Hierarchies

Makes Dijkstra faster and still **correct**

Pre-calculation:

- Introduce node ordering
- Create shortcuts to avoid unimportant nodes
- Special “upwards“ bidirectional Dijkstra while querying
- Recursively unpack shortcuts to get edges → Path

Limitations:

- Uses a lot more RAM
- Every profiles (fastest, shortest, ...) needs a pre-calculation, cannot be done on-demand

Numbers

World wide

- For car: ~120 mio edges, ~100 mio nodes
- Takes ~1h to import and requires 20GB RAM
  or less if mem. mapped config, but then use SSD!
  To run this 9GB are required

With enabled Contraction Hierarchies

- preparation takes ~2h (cars) and requires 24GB
  to run this 16GB are required
- Moscow-Madrid is under 0.04s instead >10s
- Compared to the fastest commercial Maps APIs:
  - for embedded or in-LAN queries it is ~5x faster
  - for calls over http it is similar fast

Links

- graphhopper.com
- graphhopper.com/maps
- graphhopper.com/#community
- github.com/graphhopper

Thanks!