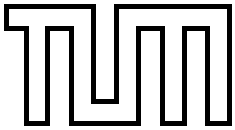
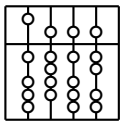


Technische Universität München
Fakultät für Informatik



Interdisciplinary system Development Project (IDP)

Reach Graphs

Florian Detig
detig@andlabs.eu

Aufgabensteller: Prof. Martin Brokate, Ph.D. Martin Sachenbacher

Betreuer: Ph.D. Martin Sachenbacher

Abgabedatum: 19. April 2012

Abstract

Mobility is at an interesting evolutionary turning point, transitioning from *owning* personal equipment towards *access* to custom-tailored mobility services. Transportation infrastructure overload and impeding fossil fuel shortages call for a more intelligent utilization of readily available capacities and resources. The recent proliferation of ubiquitous mobile communication technologies make a demand-responsive coordination of travel activities possible for the first time.

The IDP at hand revisits the well known *shortest path* problem as an essential subroutine in any algorithm dealing with such dynamic real-time scenarios. Some popular speed-up techniques are re-evaluated together with reasoning about why these approaches actually work so well in graphs of road networks. *Reach Graphs* are proposed as an alternative way to compute shortest paths, by directly connecting vertices of increasing reach by overlay shortcut edges, resulting in a relatively sparse graph where bidirectional routing is still exact. Beside an obvious straight-forward algorithm to construct *Reach Graphs*, also an iterative functional-style MapReduce approach is sketched that distributes the precomputation work load over clusters of machines, CPUs and hard discs.

Although not quite as efficient as previous approaches, Reach Graphs may contribute to a better understanding of the characteristic properties of a graph that make speed-up techniques actually work and makes them perform so well.

Contents

1 INTRODUCTION

Mobility is the lifeblood of economy and a key necessity for participation in society. The ability to move anywhere at any time correlates to freedom and independence as it provides for social relations and access to work, education, goods and services.

1.1 Motivation

Automobile transport accelerated heavily throughout the twentieth century and has become the dominant means of mass transportation. But this form of individual mobility is increasingly under challenge as it is reaching its limits. For the first time half of the planets population is living in cities where urban networks are severely congested with no room left for further extension of road capacities. On the other hand public transportation is of poor quality and not economically viable in sparsely populated rural areas, since a critical mass of passengers is not achieved. Furthermore an imminent shortage of fossil fuel supplies is threatening the global economy as oil exploration peaks and carbon dioxide emissions are polluting the atmosphere resulting in a runaway climate change. A recent study about mobility puts it eloquently: "*Existing mobility systems are close to breakdown.*"[27]

1.1.1 shifting the cultural mobility paradigm from "own" to "use"

However a possibly more sustainable future of mobility is already on the horizon. The status symbol of the "*car*" is gradually shifting towards the "*mobile phone*". Owning and maintaining personal transport infrastructure becomes less attractive for a growing demographic with lifestyles that make private cars less worthwhile. Ad-hoc car-rental schemes are filling this niche by decoupling *ownership* from *usage* and making the automobile "freedom to move" accessible with the press of a button. Car manufacturer acknowledge this trend by transitioning towards service providers, not selling "*automobiles*" anymore, but selling "*automobility*" to pay by the minute.

This gradually happening shift in mobility culture yields different travel patterns with generally more efficient transport logistics and a more beneficial modal split. While personal vehicles tend to be used as much as possible to make up for fix costs, a more deliberate modal choice is incentivized by collective infrastructure pools. E.g cars are only chosen in cases when they are really the best suited mode of transport, not just because they need to pay off. Public shared transport on the other hand is strongly encouraged since it yields the highest common and personal overall benefits. Thus empty traveling seat space and underutilized parking vehicles are minimized and the overall efficiency of the transportation logistics as a whole gets improved.

With all this spontaneous flexibility and unbound mobility power available, the user interface is always the same: a familiar button on the mobile phone to press FIRST and THEN make a well informed decision on how "best" to get somewhere.

Figure 1: demand-responsive cooperatively organized electrified mobility as a service

1.2 E-Mobility

The electrification of transport is another promising direction already visible in china where pedal electric vehicles witness a renaissance of (motorized) bike travel. Although electric power as well as hydro are just energy *storage* technologies and not energy *sources* that could make up for shrinking oil supplies, there is still potential as some day sufficient electric power might be retrieved from regenerative sources. Battery constraints constitute the technological bottleneck at the time of this writing resulting in short driving ranges, long charging times and comparatively high costs. Hence collaborative usage patterns appear even more compelling for electric mobility.

1.2.1 mobility goes mobile: the cell phone as the “way“ to move

The recent proliferation of mobile phones has made it possible for the first time to capture the transport demand and the supply in real-time just as it happens. Organizing travel activity is essentially a communication problem and as such it is addressable by communication technology. “Smart“ phones are predestined to become *THE* primary tools to coordinate any movement and how to get around. By enhancing communication between always-on(line) connected traffic participants, transport can be organized in a more streamlined, flexible and demand-driven way.

1.3 Challenges

Within a completely dynamified transport scenario, challenges become apparent. Coordinating cooperatively operated fleets of electric vehicles requires algorithms that adequately match resources, given that the situation is continuously changing. Constant updates and unpredictable ad-hoc variations make planning rather hard.

A fundamental subroutine in any transport logistics optimization application is finding shortest paths in graphs, a classical showcase of algorithmic engineering. Specifically the application of computing driving directions in road network graphs did receive a plenitude of research and is widely regarded as essentially solved. Fast speed-up techniques exist that are exact and feasible in practice [29]. However tracking large fleets and integrating multiple modes of transport in a real-time setting requires efficient updates as well as fast query times for many concurrent calculations. Moreover many algorithms need to run on battery-powered mobile equipment with lower processing power, tight battery constraints and a much slower main memory. The IDP at hand revisits the shortest path problem in the light of this perspective.

1.4 Notation

The road network is modeled as a graph $G = (V, E)$ where V is a set of vertices, representing junctions that are connected by a set E of edges $(u, w) : u, w \in V$, representing road segments between the junctions. The weight function $w(e) : e \in E$ is a distance metric like travel-time, fuel consumption or euclidean distance. $w((u, w))$ is abbreviated as $w(u, w)$. A path $P = \langle v_1, v_2, \ldots, v_n \rangle \in V$ is a sequence of neighboring vertices with $(v_i, v_{i+1}) \in E$. The length $c(P)$ is the sum of all the weights of the edges in P, i.e.: $c(P) = \sum_{i=1}^{n-1} w(v_i, v_{i+1})$. $\sigma(s, t)$ denotes the length of a shortest path in G from a start vertex s to a target vertex t. Let $AP(s, t)$ denote the set of all possible simple paths from s to t, then $\sigma(s, t) = \min_{p \in AP} c(p)$. In sake of simplicity, only undirected graphs are considered. However, the methods can "easily" be generalized to directed graphs.

1.5 Outline

The IDP at hand is organized as follows: After this introductory motivation, some preliminary fundamentals are covered in section 2 about related work. Specifically techniques such as *Reach-Based Routing* and *Contraction Hierarchies* are examined in detail. The notion of *reach-graphs* is formally introduced in section 3 that constitutes the core of this work. In section 4 some experimental results are presented and section 5 concludes with some final remarks and an outlook on further research.

2 RELATED WORK

Road networks expose some distinguishing characteristics compared to general graphs. E.g. they are almost planar and usually sparse, i.e. the average degree of vertices is relatively low. Another property is that they allow effective goal directed search using euclidean distance heuristics and triangle inequality to shrink the search space. Road infrastructure has been build intentionally in a way to minimize construction costs while maximizing connectedness, i.e. fast travel times between any location. This results in a natural hierarchy of importance among the different types of streets. Most shortest paths use small low-level roads only locally close to source or target. By removing these edges, and sufficiently far away, only considering high-level edges, the overall search space can be reduced significantly without sacrificing exactness.

A plethora of special purpose algorithms and dedicated optimizations has been proposed to speed up the computation of driving directions in street/road networks. Most of these approaches "outsource" much of the work to a preprocessing step and (re-)use it to accelerate an arbitrary number of subsequent shortest path queries. Auxiliary data, like additional edges or labels, is (pre-)computed in advance and then used to sparsify the search space during query execution by pruning and directing. In this manner, fewer nodes and edges need to be inspected, speeding up the algorithm. In some sense such techniques can be seen as pre-computing parts of shortest paths and reusing that work by joining just a small number of these (sub-)paths. Obviously this represent an adjustable trade-off between the preprocessing and query costs. An extreme case would be to pre-compute all paths, allowing constant time look-ups. On the other side of the spectrum, all work is done at query runtime without any help. For larger graphs the former requires prohibitive space and time, since e.g. Northern America or Western Europe alone already have about 20 million vertices. The latter would simply be an algorithm running on the original unmodified graph.

Many variations of this basic idea exist in the literature, among which, *Contraction Hierarchies*(2.2) and *Reach-Based Routing* (2.3) are examined in this section. The latter inspired much follow-up work and a recent theoretical understanding [13] which formalizes the distinctive properties of road network graphs that explain why the empirical results actually work so well in practice (see *Highway Dimension* 2.4). The former is a powerful simplification of other hierarchical speed-up techniques, like e.g. *Overlay Graphs* [28], *Highway Hierarchies (HH)* [31, 32], *Highway-Node Routing (HNR)* [30] or *Transit Node Routing (TNR)* [26]. Furthermore it constitutes the base for the latest (even faster) approaches, the *SHARC* [15] algorithm and the *hub-based labeling* algorithm (HL) [12], which are more complicated and have much higher space requirements. The notion of *Reach-Graphs* (introduced in section 3) is essentially a combination of both techniques. But first Dijkstra's Algorithm is reviewed on the next pages, to introduce terminology and settle common ground.

2.1 Dijkstra's Algorithm

Dijkstra's Algorithm [19] is a basic template and subroutine for many graph search algorithms. It solves the single-source shortest path problem (SSSP) exactly by iteratively growing a sphere-like shortest-path tree (SPT) from one source vertex to all other vertices in the graph. Initially all nodes get assigned the label *unvisited* and tentative distances are set to infinity. A priority queue holds the *search frontier* which is the border of the sphere-like search space. These nodes constitute the leaves of the growing SPT and are labeled as *relaxed* vertices. There is some path discovered to them but it is not known if it is the shortest one possible. Initially only the source vertex s is inserted into the queue with a tentative distance of zero as the priority. In each iteration the *relaxed* vertex with the smallest distance is removed from the queue. It is labeled *settled* and the tentative distances of its neighbors are updated by relaxing its outgoing edges. A shortest path to this *settled* node is now exactly known as the tentative distance becomes final. No shorter path can be discovered, since all other nodes in the frontier are at least that far away. In the relaxation step all not yet *settled* neighbors are (re-)inserted into the priority queue and their tentative distances updated with a possibly shorter path via the just *settled* node. A pointer to the predecessor node is maintained as parent in the SPT to later reconstruct the path. In this manner, all nodes are gradually settled in ascending distance order from the source. The algorithm terminates once all vertices are *settled*, or once a desired destination is *settled* in case of a single-pair shortest path problem (SPSP). A shortest path is obtained by following the parent pointers in the SPT from t until the root node s is reached. See figure 13 for a code example.

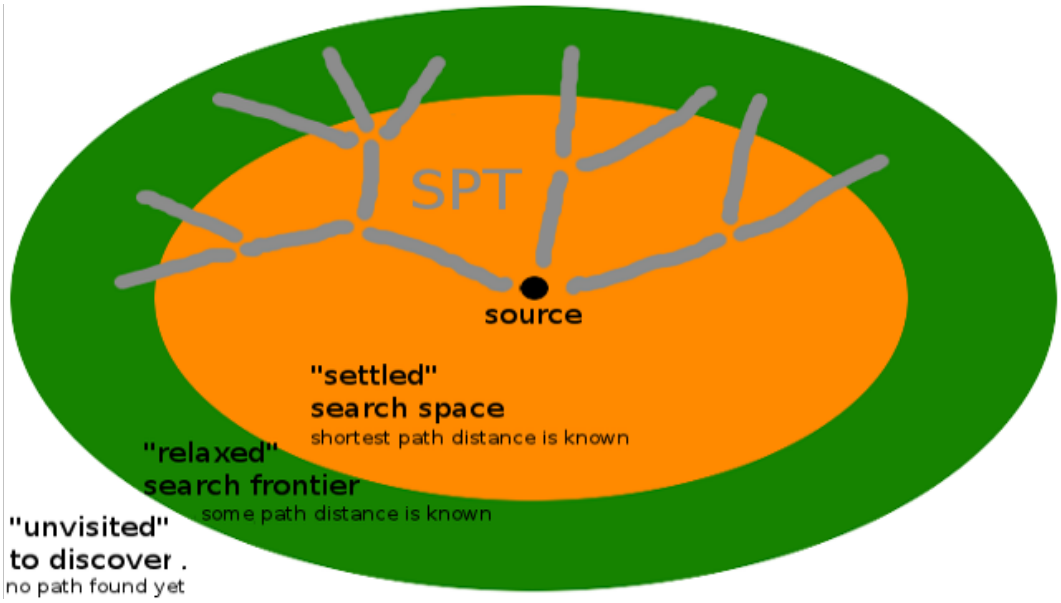


Figure 2: sphere-like expanding shortest path tree

In the worst case Dijkstra's Algorithm solves the SSSP in $(m+n)$ priority queue operations. This results in a running time complexity of $O((n+m)log(n))$ for an ordinary binary heap or $O(nlog(n)+m)$ if e.g. a Fibonacci heap is used.

Bidirectional Search Since its publication in 1959, Dijkstra's Algorithm has been subject to many improvements. More sophisticated techniques have been developed, as the algorithm is rather slow on large graphs. Bidirectional Search brings a constant factor speedup and is the basis for many techniques. As the name implies, it searches in two directions simultaneously until the frontiers meet. One SPT grows forwards from s towards t and one SPT grows backwards from t towards s. The two searches are interleaved and run in tandem. They stop once the search frontiers meet in between and the reverse of the backward path is then concatenated to the forward path. Once a node in the middle is settled by both searches, a first path is found, but this is not necessarily the shortest one possible. The algorithm keeps track of a tentative shortest path P and continues until all keys in the priority queues are greater than the length of this path P. It can terminate as soon as $min(forwardQueue.getMin(), backwardQueue.getMin()) > c(P)$. Every path that could be discovered later must be longer, because of the ascending distances.

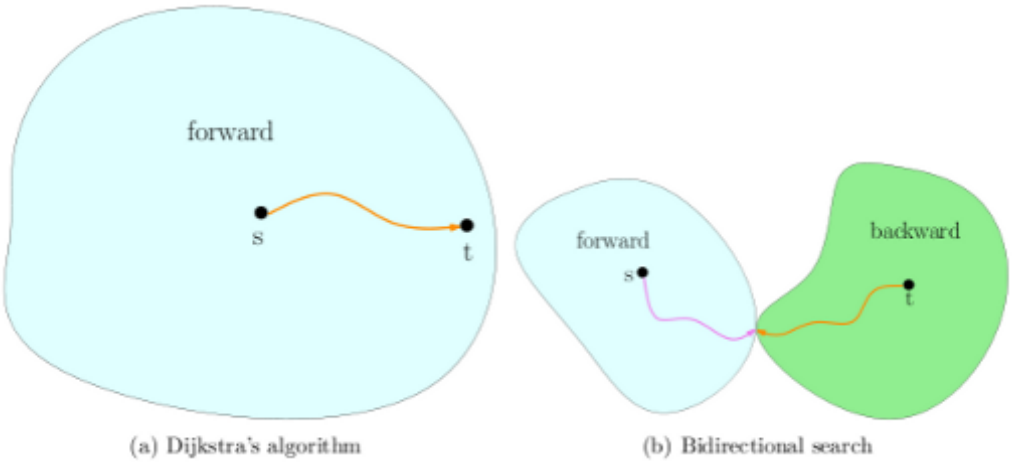


Figure 3: running the algorithm bidirectionally leads to smaller search space(s)

The advantage of bidirectional search is that the search space is reduced. This results in better performance, as less nodes are touched and the priority queues stay smaller. Intuitively two circles of half the radius cover less area than one full radius circle. Assuming that both SPTs have a branching factor b and the shortest path contains d nodes, then each of the two searches has a time complexity of $O(b^{d/2})$. As an intersection of the search spaces can be identified in constant time by hashing, the sum in this simplified model of complexity is much smaller than a single unidirectional search that would run in $O(b^d)$.

2.2 Contraction Hierarchies

One of the most effective SPSP speedup techniques is contraction hierarchies (CH). Conceived by Robert Geisberger et al. in 2008 [20], it is now widely used in practice. Among others it powers the routing engine on the official open street map (OSM) website, the web services of cloudmade [2] and mapquest [6] and is said to be used by google maps. There are countless open source implementations out there, like e.g. Graphserver [4], Opentripplaner [9] or OSRM[8]. CH is a powerful simplification of previous hierarchical routing approaches and until today provides the best ratio between space and time consumption. It is a remarkably simple technique that is surprisingly effective on graphs of street networks. E.g. on the European network, random queries visit fewer than 500 vertices on average [13]. Preprocessing takes about 10 min and adds fewer shortcuts than there are edges in the original graph.

2.2.1 Node Contraction

The core concepts of CH are *shortcuts* and *contraction*, both happening in the preprocessing stage. Contraction simply means removing nodes from the graph and introducing shortcut edges that bypass the contracted nodes to preserve shortest path distances in the remaining graph. Consider a vertex u to be contracted (fig 4). Local searches compute shortest paths between all its neighbors and if such a path P contains u as an intermediate node, a new (shortcut) edge with $\sigma(P)$ as its weight is inserted that directly connects the neighbors. Note that this is not necessary if a so called *witness* path is found, which is a shortest path without u in it. Throughout the preprocessing all vertices of the graph are contracted sequentially until none is left. This total order defines a hierarchy of ever fewer nodes and ever longer edges. The resulting hierarchical graph is called a *contraction hierarchy*.

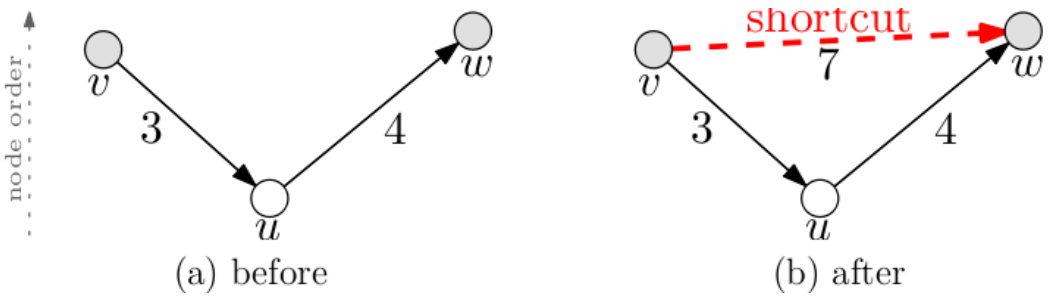


Figure 4: contracting node u and introducing one bypassing shortcut edge (v,w)

2.2.2 Query Algorithm

The SPSP query runs as a bidirectional Dijkstra's Algorithm that only goes upwards, i.e., it skips low-level nodes and only relaxes edges leading to nodes higher up in the hierarchy that have been contracted later. Whenever a search would have to go downwards to reach a node, the relevant part of the path is represented by a shortcut. As the algorithm progresses, only few high-level nodes with long overlay edges are inspected, which leads to a significantly smaller search space. Forward and backward search frontiers eventually meet somewhere in between at the most "important" node on a shortest path.

2.2.3 Edge Difference

Queries will always be correct, irrespective of the order in which the nodes are contracted. However node ordering is crucial for query time and the size of the precomputed hierarchy, since contraction decreases the steps that a query needs to perform but shortcuts increase the graph size. Intuitively nodes that appear on few shortest paths should be contracted first and placed low in the hierarchy, while nodes that appear on a lot of paths should be contracted last. A heuristic estimates the "importance" of each node, i.e. its attractiveness to be contracted next, because the computation of an optimal — search space minimal — ordering is supposedly NP-hard [14].

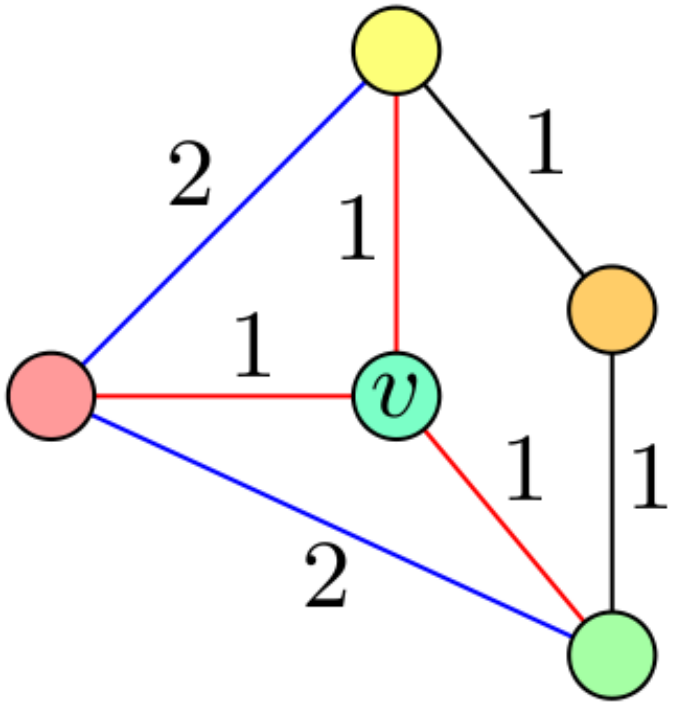


Figure 5: Edge Difference of a vertex v with one witness path: $2 - 3 = -1$

The importance function is a linear combination of several terms, such as the node's current degree and the hierarchical levels of its already contracted neighbors to ensure a uniform distribution. The central importance measure is the so called *edge difference* of a node v, which is the net difference in the number of overall edges that a contraction of v would make in the graph. It is defined as the number of newly inserted shortcut-edges that are needed to preserve path distances minus the number of edges that can be removed together with v if it is to be contracted.

Edge difference is a powerful concept that quantifies the importance of nodes. Central "traffic hubs" that are part of many shortest paths will have a large edge difference. Contracting nodes with a small edge difference first, leads to a sparse graph with fast queries. The heuristics are re-computed as the priorities change along with the sequential contraction. This happens in a lazy fashion as updates constitute the main amount of preprocessing work and are several times slower than the contraction itself. For more details on CH see [22].

Abraham et. al. observed [13] that the heuristic used to choose the next vertex to contract appears to work well at the beginning of the algorithm, when choosing unimportant vertices. However, it works poorly near the end of preprocessing, when it must order important vertices relative to one another. Finding better heuristics for node ordering is an interesting research topic. In section 4 a different approach is proposed, contracting the graph in parallel from the bottom up using a functional-style MapReduce programming paradigm.

2.3 Reach-Based Routing

Another interesting measure to identify important nodes is the concept of *reach*. Introduced by Ron Gutman [25] in 2004, *reach-based routing* inspired much of the later work, even though the computation of exact reach values has not been practical for large graphs.

The notion of "reach"

Intuitively, the *reach* of a node v encodes the size of the area where v is relevant as an intermediate node in shortest paths. To have a high *reach* value, a vertex must lie on shortest paths that extend long distances in both directions. Thus e.g. highways usually have high reach and local intersections tend to have low reach.

Given a path P from s to t and a vertex v on P. The reach of v with respect to P is defined as the length of the shorter of the sub-paths dividing P at v, i.e. $r(v, P) = min(\sigma(s, v), \sigma(v, t))$. The *reach* of a vertex v, denoted by r(v), is the maximum reach w.r.t. all shortest paths that contain v. Let SP be that set of all possible shortest paths through v. Then $r(v) = max_{P \in SP}(r(v, P)))$. This definition assumes *canonical* shortest paths to be unique which is assured by edge perturbation.

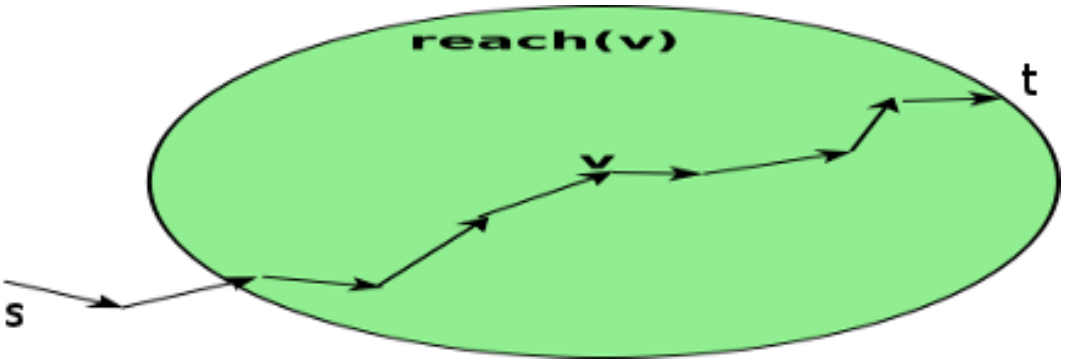


Figure 6: the reach of node v

As most nodes have a small reach, they can be ignored sufficiently far away from s and t. The original unidirectional query algorithm uses a lower bound estimate for the distance to t and the tentative distances from s to decide if a vertex needs to be considered in the search or can safely be pruned because its reach is too small to contribute to a path that long. Only upper bounds for reach are used since the computation of exact reach values would require the processing of all possible pairs of shortest paths which is still impractical for large graphs.

The original query algorithm [25] does not relax nodes that do not contribute to any path long enough to be of use. Goldberg et al. strengthened this approach by introducing a bidirectional variant (RE) [23] and a combination with A* and triangle inequality via landmarks, resulting in the REAL algorithm [24]. But their experimental results were later superseded with the advent of CH.

2.4 Highway Dimension

While these techniques do not hold any asymptotic advantage over Dijkstra's Algorithm, they are an amazing improvement in practice, touching just a few hundred vertices to answer queries in continental-scale road networks with tens of millions of vertices almost instantly with preprocessing times and space overhead that is practical for many real-world applications. However until recently these excellent results were purely experimental with no theoretically provable guarantees. Empiric evidence showed that they perform particularly well on road networks (for which they have been designed in the first place) but not necessarily on arbitrary graphs.

A theoretical understanding has only been proposed recently [13]. Abraham et al. introduced the notion of *highway dimension* to formally characterize the propertie(s) of road networks that make speed-up techniques actually work. The assumption is that these types of graphs have a low (intrinsic) *highway dimension*, due to the design intent to keep construction costs down while still providing "fast pathways". The concept of *highway dimension* is defined in terms of *shortest path cover (SPC)*.

Shortest Path Cover (r,k)-SPC) Given a directed weighted Graph $G = (E, V)$, a set $C \subseteq V$ is an (r, k)-SPC of G *if and only if* $\forall u \in V, |v \in C : \sigma(u, v) < 2r| \leq k$ and $\forall$ shortest path $P : r < |P| \leq 2r, P \cap C \neq \emptyset$.

Intuitively an (r, k)-SPC is a locally sparse set of vertices C that covers all paths of a certain length range. It is inspired by ϵ-*nets* that ensure coverage as well as separation. Every path of length between r and $2r$ contains at least one node that belongs to C and around every vertex u in the graph there are at most h nodes within a distance of $2r$. The *highway dimension* of a graph is then simply the smallest h, s.t. a (r, h)-SPC exist for all $r > 0 : r \in \mathrm{N}$.

Highway dimension is somehow related to *doubling dimension*. A graph is said to be α doubling — or to have doubling dimension $log(\alpha)$ — if every ball can be covered by at most α balls of half the radius. But a notion stronger than doubling dimension might still be necessary to fully explain the success of speed-up techniques, E.g. planar grids do not *contract* that well, even though this class of graphs has a low doubling dimension.

The authors show that a low highway dimension formally guarantees good query performance for algorithms like *contraction hierarchies* or *reach based routing* and thereby give an explanation of what might be the reason for their extraordinary good performance on road network graphs. Unfortunately the proposed polynomial-time approximation algorithm for the computation of SPC is still to slow for continental size networks. For more details on *highway dimension* see [13].

3 REACH GRAPHS

An interesting question is if for a bounded size there exists some kind of "optimal" augmented graph that can be retrieved from the original graph, s.t. arbitrary SPSP-queries take a minimum number of steps to complete. In a real-world application certain pairs of origins and destinations are queried more frequently than others. Therefore real usage statistics would be used to identify which nodes or sub-routes are "really" the most important ones to accelerate (or cache) typical queries. But beside such pragmatic reasoning a graph of low highway dimension might by itself expose some topological structure to construct some augmented auxiliary graph of a certain bounded size that would speed up all possible s-t-queries evenly well. Another interesting question would be if such an "optimal" graph can be computed (or approximated) in practice and how such algorithms would work.

This section introduces *reach graphs* which are essentially a combination of the techniques covered in section 2. First the general idea is formally introduced together with a naive implementation to construct reach-graphs. Then a different approach towards preprocessing large graphs in parallel is drafted, using a functional-style MapReduce programming paradigm.

3.1 Definition

Assumed that exact reach values are known for all vertices in a graph G, then a *reach-graph* is simply the original set of vertices V connected by a set of shortcut edges E^* that always lead to vertices of higher reach and in between have only lower reach vertices on the shortest paths they represent. In other words every vertex u has direct edges to all its nearest neighbors w that have a reach value higher than that of u. The weight of these edges is the length of the shortest path from u to w and all vertices v that lie between u and w have a lower reach value than that of u.

Definition 1. *Given a graph* $G = (V, E)$*, then a* reach-graph *is* $G^* = (V, E^*)$ *where* $E^* = ((u, w) : r(u) < r(w)$ *and* $\forall v \in shortestpath(u, w) : r(v) < r(u))$*, and* $\forall (u, v) \in E^* : c(u, v) = \sigma(u, v)$

Obviously the highest reach node has no outgoing edges, but many incoming edges from lower reach nodes around. It typically lies somewhere in the middle of the graph while nodes on the outer boundary have edges leading towards the middle.

The query runs as plain bidirectional Dijkstra's Algorithm on this reach-graph and solves the SPSP exactly by only inspecting vertices in an ascending reach order.

3.2 Correctness

Theorem 1. *The algorithm always terminates and does find some shortest path(s).*

Proof. Let vertex v be the vertex of highest reach on a shortest path from s to t. (As a special case v could be s or t itself.) Only one side of the symmetrical algorithm will be considered in this proof. The other side is analogous with s substituted by t. If all vertices between s and v have a smaller reach than s, then there exists a direct edge connecting s and v, due to the definition of reach-graphs. Otherwise there exists at least one vertex u between s and v with $r(s) < r(u) < r(v)$, which in turn will be either directly connected or indirectly via an intermediate reach vertex. Hence the two SPT will meet and thereby discover one possible shortest path. □

Corollary 1. *The searches always meet at the highest reach node on a shortest path.*

Proof. Suppose the two SPT meet at v which is not the highest reach vertex. Then on one of the sub-paths $s - v$ and $v - t$ there exists a vertex h with $r(v) < r(h)$. No edge can lead from beyond h towards v because that would contradict the definition of reach-graphs. □

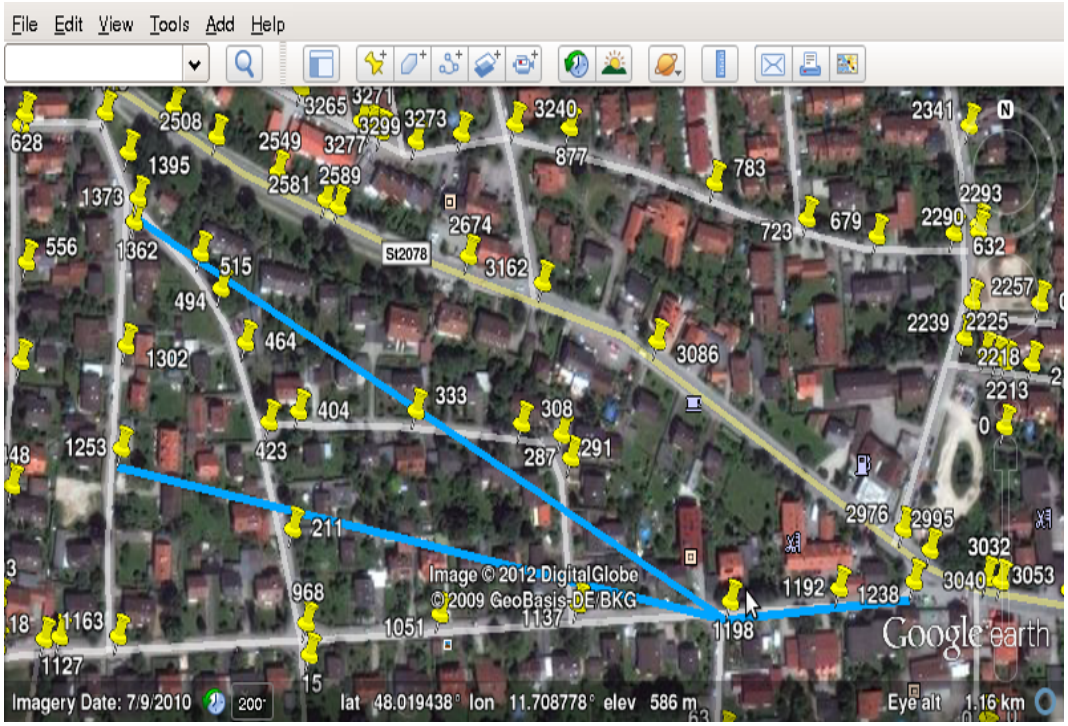


Figure 7: a vertex with three outgoing edges to its next higher reach neighbors

3.3 Construction

3.3.1 Naive Algorithm

To construct a reach-graph, reach values are needed for all vertices in the graph. A straight-forward implementation is to simply compute all pairs of shortest paths and for every vertex take the maximum over the minimum of the length of all sub-paths. A full SPT is grown from every vertex which is then traversed from the bottom up and the minimum of the distance from the root and to the bottom leaves is recorded. The maximum over all these minimums constitutes the reach value for every vertex. Canonical (unique) shortest paths are assured by simply taking the first one in case more than one shortest path exists. After computing the reach values, the SPTs are traversed again to add shortcut edges. The traversal can stop as soon as all branches are covered with an edge to a vertex with a reach value higher than that of the root vertex. On a typical workstation this algorithm is practical for small graphs.

3.3.2 MapReduce Algorithm

For large graphs like the European road network a parallelized implementation can distribute the computation work over several CPU cores and/or several machines. Multithreading is the typical way of doing parallel programming, but is error-prone, due to the complexity to synchronize the access of each thread to any shared state. A fundamental concept of functional programming is to completely eliminate such hidden dependencies. Data is explicitly passed between functions as immutable values which can only be changed by the active function at that moment. Hence functions can run anywhere independent of each other and the process of work load distribution as well as fault tolerance can be made transparent to the programmer.

The MapReduce paradigm is an emerging massively parallel programming model designed for processing large volumes of data by dividing the algorithm into a set of independent tasks that run on virtualized storage and execution resources. It was originally introduced at Google [18] to compute page-ranks on large clusters of commodity servers. Today it is the powerhouse behind most “big data“ processing. Approximating the “importance“ of websites on the graph of the World Wide Web is a somehow similar problem to computing reach values in large road networks.

The MapReduce framework is inspired by functional programming constructs, specifically idioms for processing lists of data. Applications require expressing solutions in a "dataflow-centric" manner comprised of two functions: *map* and *reduce*, which transform lists of input key/value pairs into lists of output key/value-pairs. First input data is split into multiple mapper instances which execute independent of each other. The intermediate results emitted by the *mapper* are then partitioned by key and *shuffled* over the computing cluster in a distributed merge sort operation. Every *reducer* instance receives values of the same key and merges them together returning a single output value for the key. A more complex MapReduce algorithm is usually structured as multiple rounds of MapReduce passes. The framework takes care of distributing computation tasks and gracefully recovering from any failure.

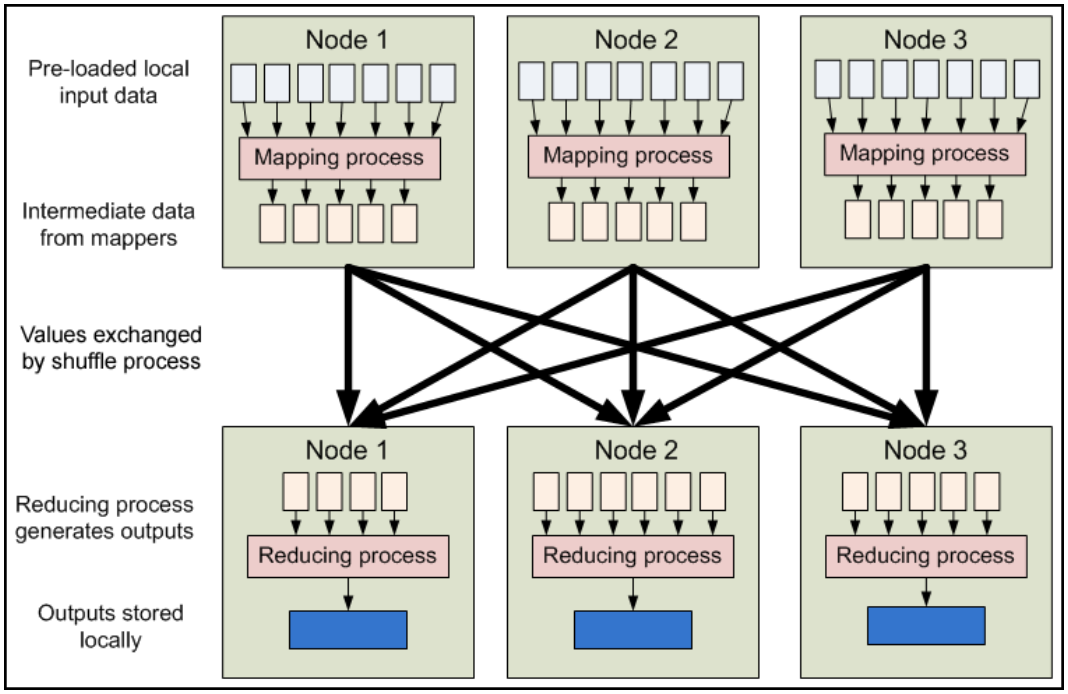


Figure 8: data flow of one MapReduce pass distributed over three machines (nodes)

The distributed algorithm to construct large reach-graphs progresses iteratively. Every iteration $i \in 0...log(n)$ is composed of two MapReduce passes and computes all shortest paths that consist of $2^i < l \leq 2^{i+1}$ edges. Consider the graph in fig. 9.

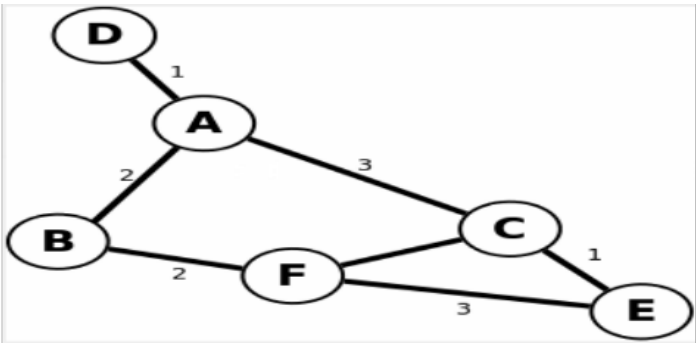


Figure 9: MapReduce example graph

The first MapReduce round gets as input a list of vertices with the vertex id as key and adjacent edges as value. The output is a list of new (shortcut) edges with a unique edge key and a list of *via vertices* as the value. The *map* function combines every pair of neighbors, and outputs a new edge that represents the path from neighbor to neighbor *via* the vertex in the middle. The key of this intermediate value is the *cantor pairing function* [16] of the vertex ids on both ends to get a unique reference to the new edge: $key_{edge}(id_1, id_2) = \frac{1}{2}(id_1 + id_2)(id_1 + id_2 + 1) + id_2$. The value is the edge length and the id of the *via vertex*. In the *shuffle* phase all edges that connect the same pair of vertices arrive at the same reducer, due to the unique cantor pairing function key. The *reduce* function takes the minimum of all edge lengths to get the actual (shortest) length of the new (shortcut) edge. Every such edge could possibly be connected by an even shorter path that consist of more than 2^i edges, but such a path can only be discovered in a later iteration.

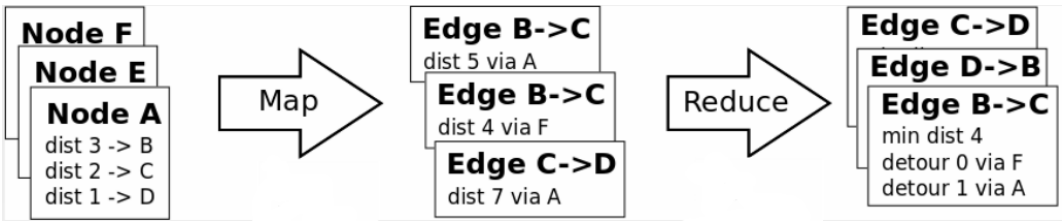


Figure 10: First MapReduce pass of one iteration

The second pass gets the previous list of edges and outputs a list of vertices. Edges that come with a *via vertex* are new and represent shortest paths that have been concatenated in the first MapReduce pass. The *map* function keys the edges by this *via vertex*. Hence the *reduce* function gets all *via edges* that go via one single vertex together with all the *direct edges* that are adjacent to this single vertex. This is where reach accumulation, edge difference computation or contraction happens. The tentative reach value is the longest *direct edge* for which a *via edge* exists. If there are no *via edges* at all then the tentative reach value becomes final due to the *optimal substructure* property of shortest paths. Longer paths that could be found in later iterations will also not contain the vertex. The edge difference is simply the number of *via edges* minus the number of direct edges. To contract a vertex, all edges that lead to that vertex are simply deleted or ignored in later iterations.

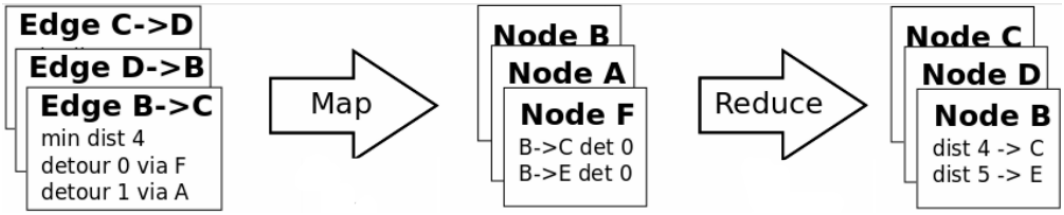


Figure 11: Second MapReduce pass of one iteration

This algorithm grows exponentially in complexity, since successively all pairs of shortest paths are included as new edges and the number of adjacent neighbors to be concatenated as new overlay (shortcut) edges becomes bigger with every iteration.

However, as soon as reach values settle and become final, the respective vertices can be ignored in later iterations since they won't contribute to longer shortest paths. Furthermore contracting vertices keeps complexity practical, since contraction can be seen as artificially restricting the reach of a vertex. It is bypassed by shortcut edges and thus becomes irrelevant for longer shortest paths. Typical road networks contain a huge number of vertices with a degree of two, which are needed for the geometric representation when drawing curved streets in visual maps. These vertices have an edge difference of zero and can be contracted already in the first iteration. Small residential roads that are only relevant for short range paths follow next, because their reach settles very soon. Together these vertices make up the majority in road network graphs, leaving only main streets and highways for later iterations.

Another benefit of this iterative MapReduce algorithm is that all possible paths are considered, i.e. not only one shortest path between every pair of vertices, but also alternative shortest paths and "almost shortest" paths with a slight detour. Hence the arbitrary restriction on canonical shortest paths can be mitigated. Paths that make a detour via certain vertices can be incorporated into estimating their importance, calculating their edge difference and help to decide which vertices to contract in every iteration. If a vertex does not lie on many shortest paths but on many "almost shortest" paths, it can still be regarded as an important traffic hub. This results in a more even distribution of reach and a better overall contraction.

4 EXPERIMENTAL WORK

4.1 Setup

For the experimental implementation *Scala* [10] is used as programming language together with *SBT* [11] as build environment. For visualization and debugging, the *Google Earth* [3] geo viewer is applied with *KML* [5] as an XML-based data format. All routing map and road network graph data is extracted from the OSM project. The *Apache Hadoop* [1] framework is used to execute MapReduce jobs (algorithms).

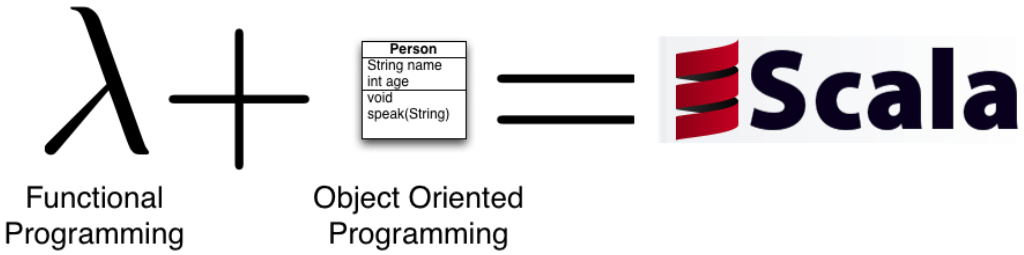


Figure 12: Scala reconciles object oriented with functional programming power

4.1.1 Scala

Scala is a relatively new language that provides a well balanced compromise between high-level expressive power and low-level machine control. It is the first language to reconcile object oriented with functional programming styles as well as offering some powerful unique language features and programming constructs, like, e.g. *implicits* or *family polymorphism*, that allow to hide technical efficiency tuning. Expressing algorithms in concise functional idioms empowers to elegantly specify *WHAT* should be done instead of *HOW* it should be accomplished technically. Hence the syntax resembles more of the succinct statements commonly found in mathematics than the verbose imperative step-by-step instructions found in typical imperative programming languages. Fig. 13 shows the infamous Dijkstra's Algorithm expressed in Scala. Despite its expressiveness Scala still has means to silently step down and tune low-level machine details, which is necessary to make processing-intensive algorithms perform (or feasible at all). Since mapping and reducing are essentially functional operations, Scala is ideally suited for writing MapReduce applications. As it compiles to Java bytecode, it is completely compatible and inter-mixable with the wide range of ready-to-go software libraries and programming frameworks available in Java.

```
while (!Q.isEmpty) {

  var node = Q.extractMin // now settled.

  if (node.id == target) { //are we allready done?
    println("PATH FOUND (searched "+spt.size+" nodes)")
    return
  }

  node.foreach_outgoing { (neighbour , weight) => // relaxation

    if (neighbour.dist > node.dist + weight) {

      neighbour.dist = node.dist + weight
      neighbour.pred = node //predecessor

      if (neighbour.visited) //before
        Q.decreaseKey(neighbour)
      else // first time seen
        Q.insert(neighbour)

    }
  }
}
```

Figure 13: Dijkstra's Algorithm expressed in Scala

4.1.2 Hadoop

As MapReduce has grown in popularity, a stack for big data processing systems has emerged, comprising layers of storage map/reduce and query (SMAQ). Such systems are typically open source, distributed, and run on commodity hardware. SMAQ systems underpin a new era of innovative data-driven products and services, in the same way that the LAMP stack was a critical enabler for the Web 2.0 era.

Hadoop is the dominant open source MapReduce implementation. Funded by Yahoo, it emerged in 2006 and, according to its creator Doug Cutting, reached web scale capability in early 2008 [17]. The Hadoop project is now hosted by Apache. It has grown into a large endeavor, with multiple sub-projects, like HDFS as distributed file system or Hive to run queries. Together they comprise a full SMAQ software stack. As Hadoop is written in Java, algorithms can as well be expressed in Scala.

Figure 14: Hadoop: massively parallel and fail-save distributed computing

4.2 Open Street Map

All road network graph data comes from OSM [7] which is a popular community mapping initiative that works like a Wikipedia for geographic information. In many parts of the world OSM already provides a higher quality than commercial map providers. But most important it is free of any restrictions regarding what is allowed to do and as such it is absolutely perfectly suited for fancy experimental research.

OSM data comes in a XML-based format that contains a lot of information that is not relevant for routing. As OSM was originally intended to render map pictures, the graph contains a lot of nodes with a degree of two, which is necessary to draw curved streets. In a first step the raw file is parsed and the road graph is extracted. Fig 15 shows the relevant Scala code to parse the XML. For Hadoop a custom *Input Format* is needed to process the data in parallel using many hard discs and CPUs.

All code written for this IDP is published under the GPL free software license and is readily available at `https://github.com/orangeman/osm_routing`.

```
print("\n -> reading nodes..")
(xml \ "node") foreach { (node) =>

    val id = (node\"@id").text.toInt
    val lat = (node\"@lat").text.toFloat
    val lon = (node\"@lon").text.toFloat
    nodes(id) = Node(lat, lon)
}
println("    ("+nodes.size+") Done.")

print(" -> reading ways...")
(xml \ "way") foreach { (way) =>

    val speed = ((way\"tag" filter { (t) => (t\"@k").text == "highway" })
                                            \ "@v").text match {
      case "secondary_link" => 30 // km/h
      case "motorway_link" => 50 // km/h
      case "living_street" => 30
      case "unclassified" => 30
      case "primary_link" => 30
      case "residential" => 20
      case "trunk_link" => 30
      case "secondary" => 60
      case "motorway" => 90
      case "tertiary" => 40
      case "service" => 30
      case "primary" => 80
      case "track" => 30
      case "trunk" => 50
      case "road" => 50
      case "path" => 10
      //case "steps" => 0
      //case "footway" => 10
      //case "cycleway" => 20
      //case "pedestrian" => 10
      //case (hw:String) if (!hw.equals("")) => println("highway "+hw); 10
      case _ => 0
    }

    if (speed > 0) {
      val ids = way\"nd" map { (nd) => ((nd\"@ref").text.toInt)}
      for ((u,v) <- ids zip ids.tail) {
        edges += Edge(u, v, dist(u,v))
        edges += Edge(v, u, dist(v,u))
      }
    }
```

Figure 15: extracting a graph from raw osm data and assigning weights to the edges

4.3 Experimental Results

The simple preprocessing algorithm (section 3.3.1) to construct Reach Graphs has been run on a few sample graphs of varying type to empirically verify its correctness and benchmark its performance and memory requirements against other techniques.

4.3.1 Reach Distribution

Fig 16 shows the distribution of reach values in a sparse rural graph with travel time as the distance metric. There are only very few (important) vertices that have a high reach value. These are typically located somewhere near the center of the graph, since this is where the longest shortest paths pass through. The vast majority of nodes has a relatively small reach, i.e. they are only relevant in a small area nearby.

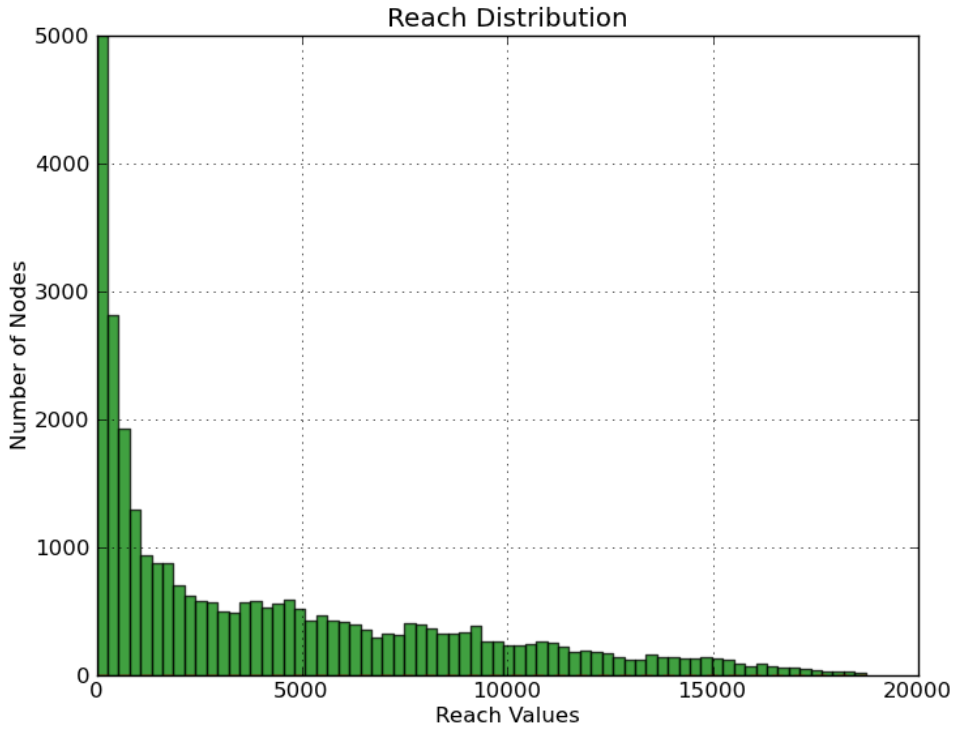


Figure 16: the vast majority of nodes has a relatively small reach

4.3.2 Query Complexity

The sizes and branching factors of shortest path trees seem a suitable indicator to measure and compare the efficiency of different techniques, since these are directly correlated to the number of priority queue operations that a query must undertake. Table 1 shows average SPT sizes for some dense urban as well as sparse rural graphs. The Contraction Hierarchies source code of the OSRM [8] project was tweaked, as this seems to be the most sophisticated among the freely available implementations.

Graph Type			Contraction Hierarchy		Reach Graph	
Density	Nodes	Edges	avg SPT size	Edges	avg SPT size	Edges
Rural	2017	4452	52	10636	129	4165
	9396	19592	49	20811	572	22390
	33241	69314	95	89830	1710	91586
Urban	1093	2314	29	3114	87	2147
	5865	19838	140	44835	613	26890

Table 1: average shortest path tree sizes and number of additional edges

The results vary a lot and almost show apparently random variation patterns. On average Reach Graphs seem to have bigger SPTs than Contraction Hierarchies. However the latter produces more additional edges during the preprocessing stage, resulting in larger space requirements for the precomputed graph to fit in memory. The number of DecreaseKey priority queue operations needed to determine a SPT does also contribute significantly to the overall runtime complexity. Furthermore it is the average sizes of SPTs at the moment when the two bidirectional search frontiers actually meet, that shows the true number of steps needed to find a shortest path. More experimental work and further evaluation criteria would be necessary to draw a more accurate picture of the performance characteristics of these techniques.

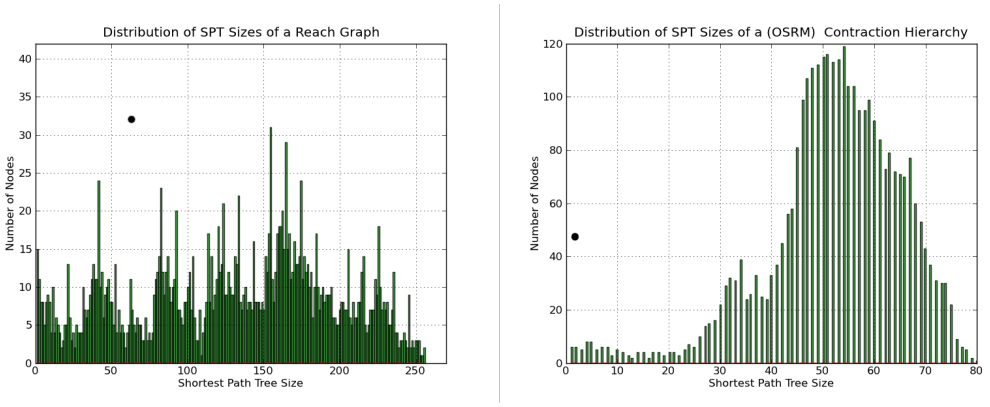


Figure 17: Reach Graphs exhibit more evenly distributed SPT sizes among the nodes

5 CONCLUSION

The last century witnessed an unprecedented automobile freedom to move anywhere at any time, that has all been fueled by the relentless exploitation of fossil resources. If this degree of personal automobility can be sustained for the generations to come, is at least questionable if not seriously doubtful. However a more sustainable future seems possible by harnessing the communication power of mobile technologies to coordinate travel activities more efficiently to make better use of available capacities.

This IDP looked at algorithmic challenges arising in demand-responsive real-time transportation scenarios. Finding multi-modal shortest paths across various modes is the basic building block in any sophisticated logistics optimization application. Algorithms need to be fast and memory-efficient to take into account the multitude of available options and to run on portable computers with tight battery constraints.

5.1 Future Work

Performance benchmarks on a shared codebase would be an obvious next step to yield more accurate results. Relating *Reach Graphs* to *highway dimension* might lead to theoretically provable complexity bounds and an explanation of why it works at all. Incorporating the notion of SPC into the distributed MapReduce algorithm could refine the strategies to decide which vertices to keep or ignore at each level.

Moreover the influence that different strategies for contraction have on the reach distribution could further enhance understanding. Contracting a node is essentially just restricting its reach artificially by introducing new edges that bypass the node and make it irrelevant for longer shortest paths. Hence combining Contraction Hierarchies with Reach Graphs looks promising for further research, since fig 17 suggests that contraction could lead to smaller less evenly distributed SPT sizes. Abraham et. al. [12] observed that the node ordering heuristics used during the CH construction works very well at the beginning, when contracting nodes with a small edge difference. But it works rather poorly towards the end, when apparently arbitrary choosing among the remaining important nodes that significantly increase the overall number of edges in the resulting graph. If just the nodes with an edge difference smaller than some threshold would be contracted, then all those nodes with a degree of two would disappear together with other nodes that unnecessarily increase the size of Reach Graphs. After this initial contraction and cleanup phase, the remaining graph would be smaller and computing reach values more feasible.

The proposed parallel Map/Reduce algorithm is just the most simple basic idea that leaves plenty of room for variation and further experimentation. Performing the preprocessing stage from the bottom up by incrementally computing longer paths, could bear an algorithm that does contraction and reach determination all at once.

Actually running the Map/Reduce algorithm on a larger computing cluster to process a graph like the European road network might also lead to insightful results.

List of Figures

References

[1] Apache hadoop framework. http://hadoop.apache.org, 2012.

[2] Cloudmade. http://cloudmade.com/about/open-source, 2012.

[3] Google earth geo viewer. http://www.earth.google.com, 2012.

[4] Graphserver. https://github.com/bmander/graphserver, 2012.

[5] Keyhole markup language. https://developers.google.com/kml/documentation, 2012.

[6] Mapquest. http://www.mapquest.com, 2012.

[7] Open street map. http://www.openstreetmap.org, 2012.

[8] Open streetmap routing machine. http://www.project-osrm.org, 2012.

[9] Opentripplanner. http://opentripplanner.org, 2012.

[10] Scala programming language. http://www.scala-lang.org, 2012.

[11] Simple build tool. https://github.com/harrah/xsbt/wiki, 2012.

[12] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *Proceedings of the 10th international conference on Experimental algorithms*, SEA'11, pages 230–241, Berlin, Heidelberg, 2012. Springer-Verlag.

[13] I. Abraham, A. Fiat, A. V. Goldberg, and R. F. Werneck. Highway dimension, shortest paths, and provably efficient algorithms. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pages 782–793, Philadelphia, PA, USA, 2010. Society for Industrial and Applied Mathematics.

[14] R. Bauer, T. Columbus, B. Katz, M. Krug, and D. Wagner. Preprocessing speed-up techniques is hard. In T. Calamoneri and J. Daz, editors, *CIAC*, volume 6078 of *Lecture Notes in Computer Science*, pages 359–370. Springer, 2010.

[15] R. Bauer and D. Delling. Sharc: Fast and robust unidirectional routing. *J. Exp. Algorithmics*, 14:4:2.4–4:2.29, January 2010.

[16] D. Cutting. Cantor pairing function. http://en.wikipedia.org/wiki/Cantor_pairing_function#Cantor_pairing_function, 2012.

[17] D. Cutting. Hadoop: A brief history. http://research.yahoo.com/files/cutting.pdf, 2012.

[18] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, Jan. 2008.

[19] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1, 1959.

[20] R. Geisberger. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. Master's thesis, Institut für Theoretische Informatik Universität Karlsruhe (TH), July 2008.

[21] R. Geisberger. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. Master's thesis, 2008.

[22] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. *Proceedings of the 7th Workshop on Experimental Algorithms (WEA08)*, 5038:319–333, 2008.

[23] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for a: Efficient point-to-point shortest path algorithms. In *IN WORKSHOP ON ALGORITHM ENGINEERING AND EXPERIMENTS*, pages 129–143, 2006.

[24] A. V. Goldberg, H. Kaplan, and R. F. Werneck. R.f.: Better landmarks within reach. In *In The 9th DIMACS Implementation Challenge: Shortest Paths*, 2006.

[25] R. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for roas networks. *Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 100–111, 2004.

[26] D. M. Holger Bast, Stefan Funke, P. Sanders, and D. Schultes. In transit to constant time shortest-path queries in road networks. 2007.

[27] W. Lerner, S. Ali, R. Baron, A. Doyon, D. Koob, O. Korniichuk, and S. Lippautz. The future of urban mobility. Technical report, Arthur D Little future lab, Nov. 2012.

[28] D. W. Martin Holzer, Frank Schulz. Engineering multi-level overlay graphs for shortest-path queries. 2006.

[29] C. V. Peter Sanders, Dominik Schultes. Engineering fast route planning algorithms. *6th Workshop on Experimental Algorithms (WEA)*, 4525:23–36, 2007.

[30] D. S. Peter Sanders. Dynamic highway-node routing. *6th Workshop on Experimental Algorithms (WEA)*, 4525:66–79, 2007.

[31] P. Sanders and D. Schultes. Engineering highway hierarchies. 2006.

[32] D. Schultes. Fast and exact shortest path queries using highway hierarchies. Master's thesis, Institut für Theoretische Informatik Universität Karlsruhe (TH), July 2005.