Algorithm Engineering

Stefan Funke

April 7, 2025

Contents

Remarks

The following scribe notes were/are developed during the following lectures:

- summer term 2017, lectures given by Stefan Funke, Filip Krumpe, Thomas Mendel, and Martin Seybold.
- winter term 2018, lectures given by Florian Barth, Daniel Bahrdt, Stefan Funke, Thomas Mendel

Some parts have been inspired/taken from scribe notes of a similar lecture *Algorithm Engineering* taught by Sabine Storandt at the University of Würzburg and a lecture on *Information Retrieval* taught by Hannah Bast at the University of Freiburg. The scribe notes do not necessarily reflect everything that was covered in class but should rather be seen as additional source to your own write-up.

1 Algorithm Engineering – Bridging the Gap between Algorithm Theory and Applications

Classical **algorithm theory** is concerned with the mathematical analyis of well-defined problems, settling their *computational complexity*. Some problems can be proven to be undecidable or NP-complete, whereas others can be solved in polynomial time. For the latter problems, one is typically interested in bounding its *asymptotic* running time, that is, how the running time develops with increasing problem size. Similarly, one might also analyze space requirements or consumption of random bits etc. From an algorithm theorist's view, a problem is well understood if asymptotic upper and lower bound for running timeor space consumption, ... coincide.

In many **application domains**, heuristic problem solutions are very common practice. Here, one typically applies intuitive and natural strategies to solve a problem without insisting on a rigorous mathematical analysis. From a practicioner's view, a problem is solved 'well enough', if the employed algorithm produces reasonable results on most typical inputs as they occur in the concrete application domain.

Both views have their right to exist. A theoretical investigation typically yields a much deeper understanding of the problem nature; furthermore, establishing undecidability or hardness might also save considerable time and effort in search for more efficient solutions. On the other hand, often real-world problems are too complex and multi-faceted to be formalized in a concise manner. Hence heuristic solutions sometimes are the only way to go.

Still, a certain hostility between theoreticians and practicioners can be observed. Theoretical results are often bashed by practitioners for being irrelevant for real-world results – an accusation sometimes hard to refute. For example, the by now only optimal deterministic $O(n)$ algorithm for triangulating a simple polygon is considered far too complex to be implemented at all and would beat easy-to-implement $O(n \log n)$ algorithms only for problem instances much larger than probably ever encountered in an application. On the other hand, heuristic problem solving is often ill-reputed for being sloppy, dim-witted, and trivial. In fact, without a rigorous mathematical analysis it is easy to miss much better solutions to a problem (and not even knowing about them).

Algorithm Engineering can be interpreted as an attempt of bridging the gap between theory and applications. It aims at the design of algorithms which are practicable and efficient for real-world problem instances, yet allow for rigorous mathematical analysis (correctness/quality of the outcome, running time, space consumption, ...).

1.1 Shortcomings of Classical Algorithm Analysis

1.1.1 Asymptotic Running Time and Hidden Constants

Algorithm theory typically is only concerned how running time develops *with growing instance size*. More concretely, an algorithm with a better asymptotic running time might only be superior to another algorithm for ridiculously large problem instances. For example, Strassen's algorithm for multiplying a $n \times n$ matrix in $O(n^{2.807})$ really beats the naive $O(n^3)$ algorithm for $n > 100$ due to huge constants hidden in the O-notation. There are plenty of such examples, e.g., the $O(n)$ polygon triangulation algorithm by Chazelle mentioned before. For some (simple) problems like sorting, exact running time analyses which also explicitly compute the constants (no O-notation), but this often requires tedious calculations and hence is often avoided by theoreticians.

1.1.2 Simplicity

Algorithms as steted in a typical theory paper are typically phrased in terms of high-level primitives like 'sort the input', 'determine the nearest neighbor', 'construct an arrangement of hyperplanes', ' use a rayshooting data structure in $\mathbb{R}^4$ ... Each of these high-level primitives often require a separate complicated algorithm or data structure to be implemented. Probably, for most algorithms published in the algorithms community, there is no implementation and never will be. Again, the $O(n)$ polygon triangulation algorithm is a good example. For theoreticians, it might be a

great achievement to improve an approximation ratio of an algorithm from $\log n$ to $0.999 \log n$ using complicated machinery and techniques, while from a practicioner's point of view this little improvement hardly justifies a considerable complication of the code.

1.1.3 Instance-Dependency

Most theoretical analyses are considered with the worst-case that could happen, be it in terms of running time, space consumption, or approximation factor. In practice, such very bad instances might occur very rarely, and it would be interesting to derive some formal statement to capture that fact. There are already several approaches to theoretically capture this aspect by either restricting the allowed input for an algorithm (think of specialized graph algorithms that make use of additional properties of the graph like planarity, bipartiteness, ...), or considering probabilistic models (assuming that the input was randomly generated and analyzing the *expected* behaviour of an algorithm on that input).

It is striking, for example, that very hard problems like the Euclidean travelling salesperson problem in practice can be solved for instances of several thousand cities with a provable certificate of optimality. Here, NP-hardness of the TSP problem does not imply untractability in practice as one might think.

1.1.4 General Position, RealRAM-Model

Many algorithms in the literature, for example in the area of computational geometry, are stated under the assumption that one can compute exactly with real numbers. If such algorithms are implemented and real arithmetic is replaced by floating-point arithmetic, things typically go awfully wrong due to rounding errors that compromise the internal consistency of the algorithms. So it is a highly non-trivial undertaking to turn even provably correct algorithms into correct implementations. Furthermore, for a theoretical analysis it is often very convenient to make certain general position assumptions (e.g. no 4 cocircular points or no 3 colinear points in the plane). In real-world instances, this might not hold, so these cases must be treated explicitly.

1.1.5 Machine Models

Typically algorithms are analyzed with respect to a very simplified machine model to keep reasoning simple and clean. The machine models might not match the real machines where algorithm implementations are executed, so the real running times predicted by theory might be far off what is experienced in practice. Most algorithms do not take into account the effect of memory hierarchies (caches, main memory, external memory), parallelism (competing accesses to memory.

1.2 The Algorithm Engineering Cycle

Design Come up with an initial algorithm solving the problem.

Analysis Mathematical analysis of the algorithm.

Implementation Careful implementation of the algorithm.

Experimentation Evaluation of the algorithm of real-world inputs.

Typically, one realizes in the experiments that the algorithm does not behave as predicted by theory. This can be either due to a bad implementation (or error in the analysis) but is often due to an unsufficient formal characterization of the real-world input. So one tries to identify and formalize important characteristics of the input and restarts the cycle with the design phase.

2 Route Planning in Transportation Networks

Computing routes in transportation networks are prime application examples for the power of algorithms. This is due to the fact that in contrast to many other real-world questions, there are typically very compact and concise *formal* specifications of the problems to be solved and very efficient algorithms have been developed that allow for the solution of even very large instances. In the following we will have a closer look at both, route planning in road as well as public transit networks.

2.1 Route Planning in Road Networks

When modelling a road network as a graph (vertices correspond to junctions, edges to road segments) these graphs have some properties that can be exploited to allow for more efficient route planning algorithms. Important characteristics in this context are amongst others:

near-planarity: road network graphs are almost planar (can be drawn with very few crossings); this also implies that the number of edges is linear in the number of vertices (due to Euler's formula)

bounded degree: road networks hardly exhibit nodes with a degree larger than 10; this also implies a linear dependency of the number of edges from the number of vertices

In most contexts, the road network graph is directed with edge weights being non-negative values corresponding to e.g., Euclidean distance, travel time, fuel consumption, ... So abstractly speaking, we are given a graph $G(V, E, c)$ with vertex set V, edge set E and a cost function $c : E \Rightarrow \mathbb{R}_0^+$. For given source–target pair s, t we are interested in finding a path π in this graph such that $\sum_{e \in \pi} c(e)$ is minimized. We call this the *shortest path problem*, even though depending on the edge weights we might actually be computing quickest or most fuel-efficient paths.

2.1.1 Dijkstra's Algorithm

The standard algorithm for solving the shortest path problem is Dijkstra's algorithm, see Algorithm 1 (essentially as described in wikipedia and many undergraduate courses). For a given source node s it computes the distances (and paths if desired) from s to all other nodes in the graph.

Algorithm 1: Dijkstra's Algorithm

```
Data: G(V, E, c) and s ∈ V
Result: distances d[] for all nodes from s
forall v ∈ V − {s} do
    d[v] ← ∞
d[s] ← 0;
forall v ∈ V do
    PQ.insert(d[v], s)
while Q≠ ∅ do
    (dist,v) ←PQ.popMin();
    forall e((v, w) ∈ E do
        if d[w] > d[v] + c(e) then
            d[w] ← d[v] + c(e);
            PQ.decreaseKey(w, d[w])
return d[]
```

In an implementation, the following simple tweaks might have a considerable (positive) impact on efficiency:

- a compact and efficient graph representation is prerequisite for acceptable performance in practice; for static graphs, an offset-array-based representation has proven to be very performant
- if only the distance to some target node t is wanted, one can abort the computation as soon as t is popped from the priority queue (worthwhile if s-t-distance is small compared to the diameter of the network)
- there is no reason to inject nodes with distance ∞ into the priority (again, in particular if s-t-distance is small)
- and yet again, if the s-t-distance is small, processing the while loop might dominated by setting distances of all nodes to ∞; if many short range queries are to be answered, it pays of to keep a list of 'touched nodes' and only reset the distances of the nodes touched during the last invocation
- in particular for sparse graphs, it does not pay off to employ a sophisticated heap data structure like Fibonacci Heaps; a simple array-based binary heap is highly efficient and even asymptotically equally good for $m = O(n)$
- for graphs with bounded degree, it also does not pay of too maintain pointers to the nodes within the heap to allow for decreaseKey operations; instead, simply insert nodes several times (multiplicity bounded by the indegree) into the priority queue
- the impact of data locality is not to be underestimated; the graph should reside in main memory that – if possible – adjacent nodes/edges are stored in nearby memory locations; rearranging the graph accordingly can improve performance

2.2 Speed-Up Techniques

From a theoretical point of view, it seems somewhat unlikely that for the shortest path problem as stated above, a more efficient algorithm than Dijkstra's algorithm can be found, since there is an obvious $\Omega(n+m)$ lower bound and if the algorithm implicitly sorts the nodes according to the distance from the source a $\Omega(n \log n)$ lower bound (of course, there might be a completely different approach which does not implicitly sort).

In practice, a reasonable implementation of Dijkstra's algorithm with a suitable graph representation takes around 5 seconds on a country-sized road network like that of Germany ($n \approx 20$ Mio., $m \approx 40$ Mio). If web-based route planners like Google/Yahoo/Bing Maps used Dijkstra's algorithm, these companies would have to maintain huge server farms to serve the incoming requests (each of which requires a few seconds of CPU core time).

These services are only made possible by decomposing the solution to the shortest path problem in to two phases:

Precomputation: The structure of G is used to precompute some auxiliary information $\mathcal{A}$ which helps later on in the query phase. This might well take longer (minutes, hours, or even days).

Query: Using the auxiliary data from the precomputation phase, incoming queries can be answered orders of magnitudes faster than using Dijkstra.

In the following we will discuss some of these speed-up techniques.

2.2.1 Arc Flags

Arc flags require the availability of an Embedding of the graph in Euclidean space. For road networks, there are some natural embeddings on the sphere or derived from that projections like the Mercator projection. Given such an embedding in $\mathbb{R}^2$ we simply put a let's say 8×8 grid ontop. Now consider one cell $\mathcal{C}$ of these 64 grid cells: we define as *boundary nodes* all nodes in $\mathcal{C}$

which have a neighbor outside the cell. For each of these nodes, we run a reverse Dijkstra over the complete network and store for each edge of the respective shortest path tree the information that it is on a shortest path too cell $\mathcal{C}$. Having done this for all cells and respective boundary nodes, each edge e in the graph has a 64bit of additional information where the i bit says whether e lies on a shortest path to some node to the i-th cell.

If we run a Dijkstra from s to t we simply determine the cell $\mathcal{C}$ containing C and restrict the Dijkstra search to edges which have the respective bit set *as long as we have not reached cell $\mathcal{C}$ itself*. For nodes inside cell $\mathcal{C}$ we consider all edges. This approach leads to considerable speedup for long-distance queries where for large portion of the search almost only the edges of the optimal paths are considered. Yet, close to the target, straightforward Arc Flags have no benefit. This can be remedied by resorting to bidirectional search (which would require to sets of arc flags, though, one marking edges leading *to* a cell on a shortest path and one marking edges leading *from* a cell on a shortest path. Yet this does not speedup queries where source and target are in the same cell. Here a hierarchical approach helps: one first constructs arc flags for let's say a 8×8 subdivision. Then one imposes a 32×32 subdivision and creates arc flags for those as well, but only storing arc flags for a cell at edges which are at most 4 cells away (since edges further away can be dealt with by the 8×8 arc flags). This can be extended to several levels to ensure speedup both for long-distance as well as short-distance queries.

The main drawback of arc flags are their considerable space overhead as well as the construction time. If we have a road network with n nodes very regularly distributed, for a 8×8 grid we expect roughly around $14 \cdot \sqrt{n}$ boundary nodes. For each of these nodes a Dijkstra over the whole network has to be executed. For very fine subdivisions or hierarchical versions with many levels, the number of bits to store at each edge also grows rapidly. Yet, arc flags are a conceptually very simple and effective speedup technique which allows for two to three orders of magnitudes improvement compared to plain Dijkstra.

2.2.2 Reach

One of the first theoretically sound speed-up techniques was the so-called *reach* by Gutman [1]. We will deviate quite a bit from its original presentation, but you are invited to verify that the main ideas are preserved in our presentation. The idea here is to first estimate the *importance* of an edge for all edges in a graph in a preprocessing step. Intuitively, an edge is important, if it appears 'in the middle' of some long shortest path. At query time executing Dijkstra's algorithm, while working on the outgoing edges of some node far away from both source and target, we can then restrict only to 'important' edges. Only near source and target, unimportant edges have to be considered. For sake of a simpler presentation we assume that edges bear Euclidean distance (and hence beeline distance to the target is a lower bound to the shortest path distance).

More formally, consider some shortest path π from s to t and an edge $e = (v, w) \in \pi$. We define $reach_\pi(e) := \min(d(s, w), d(v, t))$ and $reach(e) := \max_\pi reach_\pi(e)$. That is, $reach(e)$ is taken as the maximum over all shortest paths within the network. Naively, one could precompute all reaches by running Dijkstra from each node (exercise: how to compute reaches exactly then?). At query time when running Dijkstra's algorithm and popping a node v from the priority queue, one has to decide whether to relax edge $e = (v, w)$. But with the reach precomputed we can ignore edge e if $reach(e) < \min(d(s, w), |vt|)$, since then it cannot be part of the shortest path from s to t. If edges do not bear Euclidean distances, running a bidirectional Dijkstra also yields an immediate upper bound for the distance from v to t (and the same reasoning for pruning edges can be applied in the backward search).

Using this approach allows a speedup of about one order of magnitude compared to plain Dijkstra. The main drawback is the infeasible (timewise) preprocessing of running n Dijkstras. Preprocessing becomes feasible, though, if one does not insist on exact reach values but *upper bounds* on the reach. Note that with upper bounds, the modified query procedure is still correct. The following scheme allows for the efficient computation of upper bounds of the reach.

Estimating the importance of edges In the following we explain our way of computing levels or reaches for the edges in the graph. Let s, t be nodes in G, $\pi(s,t) = sp_1p_2 \dots p_h t$ the shortest path from s to t. We denote by $f_s(t) = p_1$ the node adjacent to s in the path $\pi(s,t)$. When computing shortest path distances from a node s we call a node v Δ-*far* if v is settled and $d(f_s(v), v) \geq \Delta$.

For given values α, Δ we say an edge $e = (p,q)$ is (α, Δ)-important if (p,q) lies on the shortest path from some node s to some other node t with $|st| \geq \Delta$, and $d(s,q), d(t,p) \geq \alpha\Delta$. Our basic procedure to compute all (α, Δ)-important edges is relatively simple and can be stated in pseudo-code as follows:

LiftEdges(G, α, Δ)

1. for all $v \in G$:
   (a) grow a shortest path tree until the predecessors of all active nodes are Δ-far.
   (b) for all edges (p,q) on a shortest path from v to some Δ-far node z, if $d(v,q) \geq \Delta\alpha$ and $d(z,p) \geq \Delta\alpha$, mark the edge (p,q)
2. inspect all edges, and lift an edge iff it was marked by at least one of the above shortest path computations

This procedure grows around every vertex a shortest path tree of radius about Δ and lifts all edges that intersect 'the middle' (as given by the parameter α) of a path of length at least Δ. It is easy to see that every edge (p,q) in G for which there exist nodes s, t with $d(s,t) \geq \Delta$ such that (p,q) lies on the shortest path from s to t and $d(s,q) \geq \Delta\alpha$ as well as $d(t,p) \geq \Delta\alpha$, gets 'lifted' by the above procedure.

Lemma 1. *LiftEdges(.) computes lifts exactly all edges that are (α, Δ)-important if $\alpha \leq 1/2$.*

Proof: Clearly all lifted edges are actually (α, Δ)-important by construction. Let $\pi(s,t)$ be a minimal $s-t$-path which witnesses the (α, Δ)-importance of an edge $e = (p,q)$. Consider the iteration of the algorithm where a shortest path tree from s is grown. Either $\pi(s,t)$ is contained in the subgraph induced by all settled nodes, then clearly (p,q) will be marked, otherwise let $\pi(s,z)$ be a maximal shortest path in the subgraph induced by all settled nodes which is a prefix of $\pi(s,t)$. We know that $d(f_s(z), z) \geq \Delta$, and hence $d(f_s(z), q) < \alpha\Delta$ since otherwise $\pi(s,t)$ would not be a minimal path witnessing the (α, Δ)-importance of (p,q). But since $d(z,p) > d(z,q) = d(z, f_s(z)) - d(f_s(z), q) > \Delta - \alpha\Delta = \Delta(1-\alpha)$, the edge will be marked by the algorithm as long as $1 - \alpha \geq \alpha \Leftrightarrow \alpha \leq 1/2$.

The idea is now to use the above procedure to filter out edges that are unimportant for long paths; intuitively, when we are about to explore a path of length more than Δ, we do not have to consider unlifted edges as long as they originate from or lead to somewhere closer than $\alpha\Delta$ to the source or target, respectively. Interestingly, we can build a hierarchy of lifted edges by reapplying the above procedure on the remaining lifted edges and a larger parameter $\Delta' > \Delta$ (let in the following M denote an upper bound for the diameter of the graph):

ComputeEdgeLevels(G)

1. $H_0 := G$
2. for $i = 1$ to $\log_2(M/\Delta)$
   (a) LiftEdges$(H_{i-1}, \alpha, \Delta\beta^{i-1})$
   (b) $H_i :=$ subgraph of H_{i-1} induced by all lifted edges from the previous call

Clearly, for any i, the respective call to LiftEdges$(H_{i-1}, \alpha, \Delta\beta^{i-1})$ computes all $(\alpha, \Delta\beta^{i-1})$-important edges for the graph H_{i-1}, but of course we are interested in what relationship those lifted edges have for the *original graph* G. Recall that we had the property that if for an edge (p,q) there exists a shortest path $\pi(s,t)$ with $|st| \geq \Delta$ such that $d(t,p), d(s,q) \geq \alpha\Delta$, the edge (p,q) was lifted to level 1 (in fact, for $i = 1$, H_1 contains exactly those and no additional edges).

Lemma 2. *H_2 contains all edges (p,q) for which there exists a shortest path $\pi(s,t)$ in G (!) with $|st| \geq \Delta\beta + 2\Delta\alpha$ and $d(s,q), d(t,p) \geq \Delta\beta\alpha + \Delta\alpha$.*

Proof: Let s' be the last node when going on $\pi(s,t)$ from s towards q where $d(s',q) \geq \Delta\beta\alpha$, likewise let t' the last node when going from t towards p where $d(t',p) \geq \Delta\beta\alpha$. Since $d(s,q), d(t,p) \geq \Delta\beta\alpha + \Delta\alpha$, all edges on $\pi(s',t')$ will be contained in H_1, and then by construction of s' and t', (p,q) is contained in H_2.

Lemma 3. *For $i \geq 2$ we have that if for an edge (p,q) there exists a shortest path $\pi(s,t)$ in G (!) with $|st| \geq \Delta(\beta^{i-1}) + 2\Delta\alpha \sum_{k=0}^{i-2} \beta^k$ such that $d(s,q), d(t,p) \geq \Delta\alpha \sum_{k=0}^{i-1} \beta^k$, then (p,q) is contained in H_i.*

Proof: Along the same line as the previous lemma.

There are several possibilities to choose β and α, we will use $\beta = 2, \alpha = 1/4$. For these parameter values we obtain the following property:

Corollary 1. *The graph H_i contains all edges (p,q) for which there exists a shortest path $\pi(s,t)$ in G with $|st| \geq \Delta 2^{i-1} + \frac{\Delta}{2}(2^{i-1} - 1)$ such that $d(s,q), d(t,p) \geq \frac{\Delta}{2}(2^{i-1} - 1)$.*

Query Routine For an edge e let $h(e)$ the maximum i such that e is contained in the graph H_i. When running Dijkstra's algorithm and considering a node v with $X = \min(d(s,v), |vt|)$, an edge $e(v,.)$ can safely be ignored if $\frac{\Delta}{2}(2^{h(e)-1} - 1) < X$.

2.2.3 A* with Landmarks

A* has been a common tweak for Dijkstra's algorithm for quite some time. Remember that unidirectional Dijkstra essentially searches in all directions from the source, not really focusing in direction 'towards' the target. A* is one way of guiding the search more towards the target. It relies on the availability of a function $\phi : V \Rightarrow \mathbb{R}^+$ which lower bounds for every $v \in V$ the distance to the target. In case of Euclidean distances on the edges according to a respective embedding, the beeline distance yields such a function. Then, for each edge (v,w) its original cost $c(e)$ is replaced (in fact modified on the fly) by $c(e) + \phi(w) - \phi(v)$. Dijkstra is then run on the graph with modified edge costs. Clearly, it will return the optimal path as the structure of shortest paths do not change (Exercise: why?). Why could that be more efficient? To see why this is beneficial, consider the (optimal) case that ϕ *exactly* returns the shortest path distances to the target. Then, all edges on the shortest path from s to t have modified edge costs of 0!, So Dijkstra will only work along these edges and reach the target.

For A* being efficient, the availability of a good function ϕ is important. For graphs with Euclidean edge costs, beeline works quite well, in case of travel times, a good function is much harder to come up with. This is where the idea of landmarks comes into play. Consider a landmark $L_1 \in V$ and define as function $\phi_1(v) := d(v, L_1) - d(t, L_1)$. Again, this always yields a lower bound to the target (Exercise: why? Hint: triangle inequality). Why could that be efficient? Consider again the optimal case where t lies on the shortest path from s to landmark L_1. Then all edges again will have cost 0 and Dijkstra's algorithm will follow those edges first and reach the target without any unnecessary work. It is unlikely, though, that one single landmarks yields good functions for all source target combinations. So the idea is to choose a *set of landmarks* $L_1, L_2, \ldots, L_k \subseteq V$ in a preprocessing step and compute and store distances from all nodes to each of these landmarks. This requires the storage of k distance values for each node of the network. Then for a query when modifying the edge costs, we use for every node the best upper bound on the distance to the target (where best means the best amongst the upper bounds induced by all landmarks). In combination with reach based routing and some shortcut technique, a speedup of three order of magnitudes can be achieved, see [2]

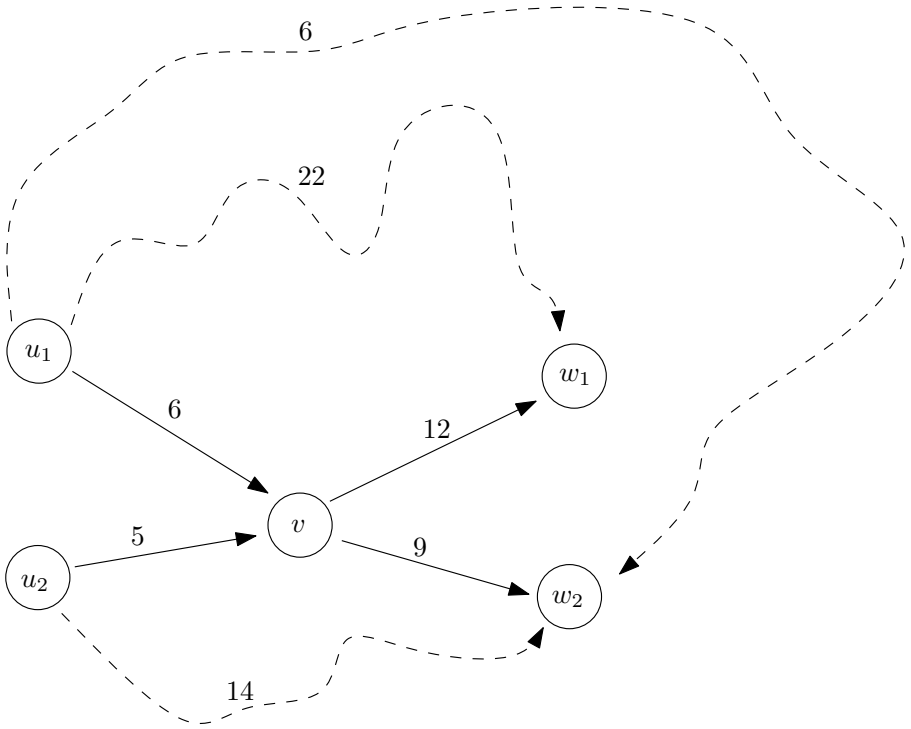


Figure 1: When contracting node v we have to create shortcuts (u_1, w_1) with cost 18 (since the shortest path from u_1 to w_1 avoiding v has cost 22) and (u_2, w_1) with cost 17 (since there is no other path from u_2 to w_1 than u_2vw_1). Shortcut (u_1, w_2) is not necessary since we can get from u_1 to w_2 on a path avoiding v cheaper than u_1vw_2, the shortcut (u_2, w_2) is not necessary since even after removal of v we can get from u_2 to w_2 at the same cost as u_2vw_2.

2.2.4 Contraction Hierarchies

In the preprocessing phase we *augment* G by additional edges E' – shortcuts representing shortest paths in the original graph G – and furthermore assign each node $v \in V$ a natural number, the $level(v)$. The result of the preprocessing phase is the function $level : V \to \mathbb{N}$ and a new graph $G^*(V, E^*, c^*)$ where $E^* = E \cup E'$ and $c^*(e) := c(e)$ if $e \in E$, otherwise $c^*(e)$ is set to the length of the respective shortest path which e represents. In practice E^* is hardly twice the size of E, that is, the space consumption of the result of the preprocessing phase is of the same order as the original graph G.

To answer a query we execute two slightly modified Dijkstra runs on G^*, one starting in s, one in t. Amongst all nodes settled from both Dijkstra runs the one where the added distances from s and to t are minimal determines the shortest path from s to t. The query is highly efficient since the modified Dijkstra runs can discard most of the nodes and edges of G^* visiting only a miniscule fraction of the graph G^*.

Preprocessing in more detail The crucial operation of the preprocessing phase is a so-called *node contraction*. Here the goal is to remove a node v from the graph G *without affecting shortest path distances between the other nodes*. To that end, it might be necessary to add shortcuts between neighbors of v.

Consider the situation in Figure 1 where node v is to be contracted. If the shortest path distance from some s to some t is affected by the removal of v it must be the case that every shortest path from s to t contains as subpath some u_ivw_j. If we add a short cut (u_i, w_j) with cost $c(u_i, v) + c(v, w_j)$ for each u_i, v, w_j for which u_ivw_j is the the only shortest path from u_i to w_j,

we can ensure that shortest path distances between any pair of nodes s, t are preserved.

In practice we can implement the contraction of node v by starting a Dijkstra run from each u_i until all w_j are settled. A shortcut from u_i to w_j of cost $c(u_i, v) + c(v, w_j)$ is created if $u_i v w_j$ is the only shortest path from u_i to w_j. The required effort for that are as many Dijkstra runs as there are neighbors u_i. If the u_i, v, and w_j are all 'nearby', the individual Dijkstra runs are fast since they can be aborted once all w_j are settled.

With node contraction as a basic building block, the preprocessing works as follows:

```
counter=1;
while |V|>1 do
select some node v from V and contract
level(v):=counter++;
od
return level() and original graph with all created shortcuts added
```

The order in which the nodes from v are contracted has great influence on the number of added shortcuts as well as the query times. We will discuss that later.

Query To answer a query from s to t we proceed as follows:

1. in G^* run Dijkstra starting in s; when pulling a node v from the priority queue, only consider edges (v, w) with $level(w) > level(v)$. This yields paths and distances $d_s(v)$ from s to some nodes $v \in V$.

2. in G^* run a 'reverse' Dijkstra[1] starting in t; when pulling a node v from the priority queue, only consider edges (w, v) with $level(w) > level(v)$. This yields paths and distances $d_t(v)$ from some nodes v to t.

3. amongst the nodes settled from both Dijkstras, the node v^* with minimal $d_s(v^*) + d_t(v^*)$ determines the result of the query

The resulting path might contain shortcut edges which of course can be recursively unpacked (each shortcut replaces two other edges by construction).

Correctness It is not clear why the path returned by the query routine is indeed a shortest path from s to t, we will argue in the following that this is really the case. To that end assume for simplicity that shortest paths are unique (can easily be enforced by symbolic perturbation) – one can also extend the argument for the case of ambiguous shortest paths in several ways.

Consider a shortest path $abcdefgh$ (black edges) from a to h as depicted in Figure 2. Here, the vertical position of a node denotes its *level*, that is, the nodes of this path were contracted in the order $b < g < a < f < h < d < c < e$. Our query routine would not find the path $abcdefgh$ since it cannot be decomposed into two parts, one with increasing levels of the nodes (to be explored from s) and one with decreasing levels of the nodes (to be explored from t). Fortunately, we can argue that the preprocessing phase has added shortcuts such that the path $abcdefgh$ has a representation (using shortcuts) which is a sequence of upward edges followed by a sequence of downward edges.

Consider the time when node b was contracted. At that time a and c were neighbors of b and (uniqueness of shortest paths!) removing b would have altered the shortest path distance from a to c, hence the shortcut (a, c) (with added costs of the edges (a, b) and (b, c)) must have been added. For the same reason, shortcuts (f, h), (c, e) and (e, h) must have been added.

Now let v^* be the node in the path $abcdefgh$ with highest level — in our example $v^* = e$. We claim that there exists a representation of the shortest path consisting of upward edges only from a to v^*. Assume otherwise, then consider the 'last' downward edge (u, v) on some path π from a to v^* and let w be the the successor of v on this path. We have $level(u) > level(v) < level(w)$.

[1] Reverse Dijkstra considers edges in the opposite direction.

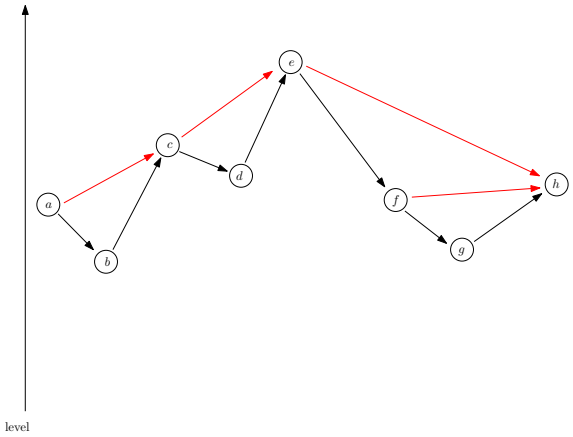


Figure 2: Correctness of the Query routine.

But then, at the time when v was contracted, the shortcut (u, w) must have been created, hence there must be also a shortest path from a to v^* exactly the same as π but without the downward edge (u, v) (but possibly a downward edge $((u, w))$. But then we apply the same argument to get rid of this downward edge as well (reducing the number of edges on the path). At some point we must end up with a path from a to v^* consisting only of upward edges (at latest when only the edge (a, v^*) is still there). The same argument applies for the downward path from v^* to t. So our preprocessing routine has ensured that a representation of the shortest path from s to t will be found.

Improved Preprocessing The efficiency of the whole approach relies on the hope that in the two Dijkstra searches in G^* only few nodes/edges have to be considered. First of all, it seems reasonable to choose a contraction order where only few shortcut edges are added. Intuitively, we are happy to contract dead-end nodes since their removal does not induce any shortcuts at all, but refrain from contracting nodes corresponding to important crossings of the road network. If we have, for example, an important 5-way crossing the contraction of the respective node might induce 25 shortcuts. So a natural strategy to pick the next node to contract in the preprocessing phase is as follows:

- compute for every node v what would happen if v was contracted, i.e. how many edges would be deleted, how many shortcuts created
- actually contract the node where the difference between # shortcuts created - # edges deleted is minimal

Unfortunately, applying only this criterion leads to some undesirable effects. To that end consider the small portion of a larger graph in Figure 3.

Applying the edge difference strategy would contract the nodes from the dead-end road one-by-one, not introducing any shortcuts, but also not speeding-up the query at all (when for example quering with the source as the end of the dead-end). Hence it seems reasonable not to contract immediately a neighbor of a recently contracted node. In fact, the original paper by Geisberger et al. proposes several strategies to implement this idea.

Recent implementations of the Contraction Hierarchy approach go a bit further and do the following:

- construct an independent set $I \subset V$
- compute the edge differences for all nodes in I

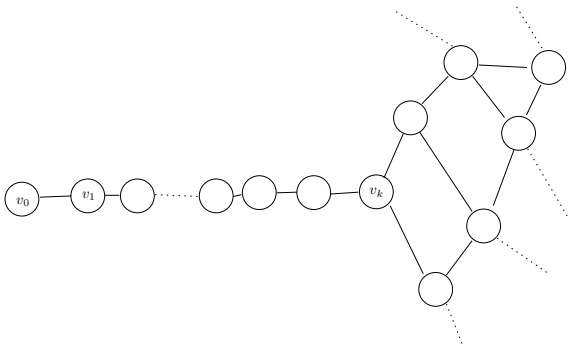


Figure 3: Bad contraction order.

- contract all nodes from I with 'low' edge difference
- assign all contracted nodes the same level counter
- increase the counter and repeat

Note that here many nodes might be assigned the same level – still, neighboring nodes are never assigned the same level. The independent set construction enforces a more uniform contraction process, still, taking the edge difference into account leads to a small number of shortcuts added. The query algorithm as well as the argumentation for correcntess does not change.

Query Optimization There are some improvements to the query process possible one of which we sketch in the following.

Probably in contrast to our intuition, the shortest path tree grown in the upwards graph rooted in s does not only contain paths that are shortest paths in the original graph (when replacing the shortcuts by the respective shortest paths in the original graph — Exercise: Give an example!). If we knew at the time when a node v is pulled from the priority queue that its distance value from s is not a shortest path distance in the original graph, we could simply discard v from consideration – and in particular do not relax any of its outgoing edges. While testing this exactly seems too expensive, there is the strategy of *stall-on-demand* which provides a simple yet effective test, which discards such nodes in many cases (look it up in the original paper). With this strategy, query times are again drastically reduced (by a factor of around 10).

Summary In practice, the preprocessing sketched above takes few minutes on the road network of Germany roughly doubling the number of edges, the queries can be answered within few millisecond (touching around 1000 nodes only in the two Dijkstras). The resulting paths contain shortcuts, of course, but these can easily be unpacked to obtain the complete path in the original graph.

A theoretical explanation for the efficiency of Contraction Hierarchies There have been several attempts to theoretically explain the overwhelming performance of CH (and other speed-up techniques). To that end, certain assumptions about the structure of the road network are made, e.g.:

- bounded *highway dimension*, [3]
- bounded *skeleton dimension*, [4]
- bounded growth, [5] and some new results (link to manuscript on webpage)

Please have a look at the manuscript linked at the webpage (only relevant are the sections dealing with CH).

2.2.5 PHAST

To compute the shortest path distances from one source node to *all* other nodes in the network it seems hard to beat Dijkstra's algorithm as it already has near-linear running time, and each node has to be touched at least once to compute its distance. Parallelization also seems difficult, even though there are some attempts, (e.g. Δ-stepping), but without convincing speed-ups in practice.

PHAST (Delling et al., 2010) is a neat way to obtain considerable speedups compared to plain Dijkstra even for single one-to-all queries. Even higher speed-ups can be achieved if distances from *several* source nodes are computed (many-to-all queries).

Again, we can expect a decent implementation of Dijkstra's algorithm to take around 5 seconds on a not too small road network (e.g., Germany $n = 20$ Mio, $m = 40$ Mio); on the other hand, linearly scanning through 60 Mio items is faster by at least one order of magnitude.

The main novelty of PHAST is to instrument a precomputed CH to (almost) linearize the memory accesses during a one-to-all shortest path computation. To compute one-to-all distances it works as follows:

1. execute a Dijkstra on the up-graph of source node s (takes few ms)
2. consider all nodes u from high to low level and set $d(u) = \min\{d(u), d(v) + c(v, u)\}$ for nodes v with $level(v) > level(u)$ and $(v, u) \in E$

This computes shortest path distances from s for all nodes in the network, which can be seen as follows: consider a node t in the network. The highest-level node on the shortest path $\pi(s, t)$ gets its correct distance already in step 1. of the algorithm. Then consider the next node v in the up-down-equivalent for $\pi(s, t)$. At the time when v is considered in step two, it will get its correct distance and so on. Ordering all edges decreasingly according to the *level of the target node*, step 2. boils down to a linear scan over the edges. This results in query times about a factor of 10 faster than Dijkstra to compute one-to-all shortest path distances ($\approx$ 200ms).

In spite of the reordering of the edges, the bottleneck of this approach is still the second step, since the accesses to node distances are somewhat non-linear, requiring frequent flushing of cache contents. This can be alleviated in case shortest paths from not only one node are desired (multiple sources). Computing distances for for $k = 16$ sources one can achieve around 100ms per tree on a single core or even 40ms in case SSE instructions are used. Parallizing on 4 cores finally yields around 20ms per tree.

This can be even taken further by executing the second step on a GPU and making use of the massive parallelism; due to memory constraints it does not make sense to compute more trees in parallel but the computation of a single tree is accelerated by parallelizing the single sweep resulting in around 5ms for a single tree and 2ms for several trees.

This finally makes an all-pairs shortest path computation which took around 200 days on a quad-core machine feasible in 94h on the same machine or 11h with GPU-support.

2.2.6 Hub Labels

While CH and PHAST still are based on some sort of traversal of (some part) the graph and hence still require substantial time (in the order of ms), Hub Labels take a quite different approach. Here the idea is to compute for every $v \in V$ a *label* $L(v)$ such that for given $s, t \in V$ the distance between s and t can be determined by just inspecting the labels $L(s)$ and $L(t)$. All the labels are determined in a preprocessing step (based on the graph G), later on, the graph G can even be thrown away.

There have been different approaches to compute such labels (even in theory); we will be concerned with labels that work well for road networks and are based on CH again. To be more concrete, the labels we are constructing have the following form:

$$L(v) = \{(w, d(v, w)) : w \in H(v)\}$$

Here we call $H(v)$ a set of *hubs* – important nodes – for v. The hubs should be chosen such that for any s and t, the shortest path from s to t intersects $L(s) \cap L(t)$.

If such label sets could be computed, the computation of the shortest distance between s and t boils down to determine the node $w \in L(s) \cap L(t)$ minimizing the summed distance. If the labels $L(.)$ are stored lexicographically sorted, this can be done in a very cache-efficient manner in time $O(|L(s)| + |L(t)|)$.

Knowing about CH, there is a natural way of computing such labels: simply run an upward Dijkstra from each node v and let the label $L(v)$ be the settled nodes with their respective distances. Clearly, this yields valid labels (as we already convinced ourselves that CH works correctly). The drawback is that the space requirement is quite large; depending on the metric and the CH construction, one can expect labels consisting of around 500 node-distance pairs. So for a graph with 20 million nodes, this boils down to a humongous space requirement. Of course, it only makes sense to keep nodes with actual shortest path distances in the label, pruning out non-optimal nodes from the labels can reduce the space to around 1/4 – around 130 labels per node. A query can then be answered in less than one *microsecond*, that is, 0.001 miliseconds!

See the original paper [6] for more details.

Hub labels are also amenable to perform distance computations in databases due to the simple query execution, there are some interesting applications sketched in [7].

Bootstrapped CH-based Hub Label Construction A very simple yet effective way of creating hub labels based on a precomputed CH is as follows: we create labels for the nodes in decreasing level, i.e., first we create the label of the node contracted last (which only has as label itself with distance 0). When considering a node v further down, we can assume that all upward neighbors of this node already have their labels constructed. We simply take the union of the labels of all the upward neighbors, always adding the distance of the edge cost to the respective neighbor. The resulting node-distance pairs are checked for optimality via the already created hub labels. This can be parallelized trivially and yields optimal (i.e. no non-shortest distances in the labels) labels.

2.2.7 Transit Nodes

For sake of simplicity, the following exposition assumes an undirected graph, even though the technique naturally generalizes to directed graphs. Similar to Hub Labels, transit nodes are not based on a graph traversal but rather a suitable lookup tables. In general, a transit node scheme is based on:

1. A set $\mathcal{T} \subset V$ of transit nodes
2. A distance table $\mathcal{D} : \mathcal{T} \times \mathcal{T} \Rightarrow \mathbb{R}_0^+$ for shortest path distances between all transit nodes
3. For every node $v \in V$ a set of *access nodes* $A(v) \subset \mathcal{T}$ together with respective shortest path distances, such that any shortest path π emanating from v and containing a transit node from $\mathbb{T}$, $A(v)$ must contain the first transit node on that path.

For given $s, t \in V$, this scheme allows to compute the shortest path distance from s to t provided the respective shortest path contains at least one transit node as follows:

$$d(s,t) = \min_{a \in A(s), b \in A(t)} d(s,a) + d(a,b) + d(b,t)$$

Hence the query consists essentially of $O(|A(s)||A(t)|)$ distance lookups. In practice $|A(.)|$ is of constant size, hardly more than 10 nodes; this concurs with the intuition that when going far away, one typically leaves the local neighborhood only on few different routes – irrespectively of exact location of the target.

Of course, there are still several open issues:

- what are 'good' transit node sets?
- how large are the access node sets?

- how can I tell for a query s, t, whether the transit node scheme is applicable at all?

In the original paper which proposed transit nodes [8], transit nodes were constructed geometrically based on a grid put over the whole map. This has the advantage of easily being able to tell whether the scheme can be applied or some fallback query processing (e.g. CH) must be invoked, but was very time consuming. It took around a day for the preprocessing, and queries were in the order of a few *microseconds*, see [8] for more details (rather easy to read).

Subsequently people have come up with much more elegant and faster schemes to determine transit nodes. We will sketch one of these schemes again based on contraction hierarchies, see [9]. It simple chooses as transit nodes the k nodes with highest level in the CH; intuitively these are 'important' nodes almost any long shortest path has go through. Choosing $k \in O(\sqrt{n})$ makes sure that the distance table $\mathcal{D}$ does not use too much memory. $\mathcal{D}$ is computed via the CH. To compute $A(v)$ for some $v \in V$ we simple run an upward CH-search from v ignoring outgoing edges from transit nodes. All transit nodes settled in this process are put into $A(v)$ together with their distance. $A(v)$ can further be pruned as well to increase efficiency.

Yet, for given s and t it is still not clear whether the transit node scheme can be applied at all. If the highest-level node on the shortest path from s to t is not amongst the top k nodes in the CH, the scheme does not apply. But this means that the CH-search spaces from s and t meet below the k top-level nodes. By storing with each node v also the set of nodes visited in an upwards search *before* hitting a transit node, we can easily check disjointness of the respective search spaces (considerably less memory required compared to hub labels). In the end, this scheme allows for query times in the order of a few microseconds (slightly slower than hub labels, but considerably less space).

2.3 Route Planning for Public Transport

The Problem of travelling via public transport differs a lot from planning routes on road networks. Even though we are able to model public transportation networks as graphs and apply many of the algorithms that are designed for road networks, they do not perform nearly as well. Additionally there are more criteria by which we evaluate the quality of a route in pulic transportation (e.g. number of transfers, cost of the journey, ...).

Notion

Because transportation networks consist of more than streets and cars running on them, we need some additional definitions:

- **Vehicle**: A physical train/bus or any other vehicle operating on the given network. E.g. the bus with chassis number 1413142.
- **Stop**: A physical place where vehicles arrive/depart. This could be a simple bus-stop or a whole train station consisting of multiple platforms. E.g. bus stop "Universität (Schleife)" or train station "Hauptbahnhof".
- **Connection**: A vehicle going from one stop to another without intermediate stops at a specific time. E.g. "S3 (heading Backnang) departs at 11:04 from Univerität and arrives at 11:10 at Schwabstraße".
- **Trip**: One specific ride of a vehicle. Defined by the vehicle, the sequence of stops and corresponding arrival- and departure-times. E.g. the train serving the stops "Herrenberg, Nufringen, Gärtringen, ..., Ötlingen, Kirchheim (T)" starting at 11:12.
- **Route**: A collection of trips using the same sequence of stops. E.g. all Trips labeled "S3".

2.3.1 The Problem

Earliest Arrival The most obvious query in public transportation would be "Starting at time t on station s, how can I reach stop s' as quickly as possible?". This is called the *Earliest Arrival Problem*. Analogous we could ask "I need to be at stop s' by time t, when do I need to start my journey at station s at the latest?". This is called the *Latest Departure Problem* (Which is basically the same problem on the reverse network).

Arrival Time vs Number of Transfers When commuting to work many people not only want to minimize the time needed to reach their destination, but also the number of changes from one vehicle to another. In this situation it's unclear what is the better choice: "Having to change once but only needing 20 minutes." or "Not changing at all but have a 30 minute ride.". We would want to report all results, which can't be directly compared and allow the users to choose their preferred route themselves.

Profile queries If we only want to travel from stop A to stop B but do not mind the exact time of departure/arrival we process a profile query: "Show me all possible journeys between 8am and 3pm from A to B (for which there is no better journey)". (Here a journey is better than another if it departs later and arrives earlier).

2.3.2 Generic Graph-Representation

There area two natural ways in which we can model public transportation networks as graphs. On these representations slight variations of Dijkstra's Algorithm can be used to navigate the user from one station to another.

Time-Expanded Model In the Time-Expanded Model we create a node for every arrival-/departure that takes place. For example there would be a node, which realizes: "S1 (heading Kirchheim) departs at 10:44 from Universität" and another node which realizes "S1 (heading Kirchheim) arrives at 10:50 at Schwabstraße". Because these are consecutive events of the same train, we introduce an edge bewtween the two nodes.

We also have to model waiting times a stop. To this extend we connect all nodes of a station ordered by time. This allows us to reach the node "S1 (heading Herrenberg) departs at 10:46 from Universität" which is connected to the node "S1 (heading Herrenberg arrives at 10:47 at Österfeld)".

Time-Dependent Model In the Time-Dependent Model we create a node for every stop in the network. We create edges between two nodes if there is a vehicle directly connecting the two respective stops. The edge contains all possible connection between the two stops. The edge between *Universität* and *Schwabstraße* would (among others) contain the following information: "S3 (heading Backnang) departs at 11:04 and arrives at 11:10", "S1 (heading Kirchheim) departs at 11:14 and arrives at 11:20", "S2 (heading Schorndorf) departs at 11:24 and arrives at 11:30", ...

See [10] for a more detailled explanation.

2.3.3 Differences to Road Networks

As stated earlier many of the algorithm for road networks do not work well when applied to public transportation networks. Here we give a quick overview of why this is the case.

Shortest-Paths do not factor into Shortest Paths One of the biggest differences between public transport and road networks is, that shortest paths do not neccessarily factor into shortest paths (depending on the model used). Consider a 4-stop journey with stops A, B, C and D using

two vehicles, which serve $A - B - C$ and $B - C - D$ respectively. Also the first vehicle arrives at B and C before the second one.

To get from A to D one needs to transfer once - either at stop B or at stop C. If the former is used the subpath $A - B - C$ uses two vehicles instead of one and arrives later than neccessary. If the latter is used the subpath $B - C - D$ uses two vehicles instead of one and departs earlier than neccessary.

Bidirectional Search The reverse search is more complicated in public transportaion networks because we would need to know the time of arrival to start the reverse search. But finding the arrival time is an essential part of most queries in the first place. One remedy is to start the reverse search from all possible time points at the target station simoultaneously. But this makes things rather complicated.

Hierarchy Public transportation networks regularly lack the hierarchy road networks offer. While for certain distances a hierarchy emerges (think about long distance trains) there is hardly any hierarchy on the lowest levels (think about intra-city journeys). This makes the local queries, needed to hop on to the top levels of the hierarchy, very expensive and thus prohibits efficient pre-processing.

Shortcuts Most stations on public transport networks are served by multiple trains/busses. This results in high node degree on most nodes which leads to a high number of shortcuts. For a sufficiently large number of shortcuts their benefit quickly vanishes.

For further differences and more detailled explanations see [11].

2.3.4 Algorithms

Dijkstra As mentioned earlier Dijkstra's Algorithm can be applied to the graph representations with only slight modifications.

In the Time-Expanded Model we start at the first node of the source stop, which has a time greater than the starting time and run a normal dijkstra. To get our result we have to find the earliest node of the target stop, which was settled.

In the Time-Dependent Model we have to compute edge costs on the fly when expanding a node (because every edge represents many connections). We could do so by doing a binary search over the edges entry to find the earliest connection we can get, considering the time we reached the current stop.

Connection-Scan One major problem of Dijkstra's Algorithm is it's mostly unstructured memory access pattern. The Connection-Scan algorithm tries to remedy this by not using a graph at all, but implement a linear sweep over the list of all connections.

The algorithm starts by creating a list of all connections sorted by increasing departure time.

For an Earliest Arrival Query the algorithm works as follows:

1. Store earliest arrival time τ_p for each stop p (initialize to ∞)
2. Consider all connections in order. A connection is *reachable* if the label τ_p of the connection's source station is less than the connection's departure time. We can than use the connection's arrival time to possibly update the label $\tau_{p'}$ for the connection's target stop p'.

While it doesn't sound overly efficient to consider *all* connections, the nearly linear memory access pattern of this algorithm allows it to beat Dijkstra's algorithm on time expanded graphs while being very easy to implement.

See [12] for the original paper.

RAPTOR The RAPTOR algorithm also tries to improve on Dijkstra's Algorithm by replacing the graph structure with another data-structure offering better memory access patterns. In fact the RAPTOR algorithm is rather intuitive. It works in rounds - in round k it computes arrival times for stops that can be reached with at most k trains/busses. The algorithm can be summarized as follows:

"Round 0": Set the time for the starting station.

The following rounds reapeat:

1. Consider all trips serving a stop whose time was updated in the last round.
2. Scan along the stops of the trip and check whether the stop was reached before the train departs. This means we could hop onto this vehicle at that stop.
3. Once we were able to hop onto a vehicle we can use this vehicle to reach the following stops. This means we can update the stop's time if this vehicle arrives earlier than the stop's current label.

The algorithm terminates when all stops have their final label.

See [13] for the original paper.

2.3.5 Considerations

The described algorithms were rather simplified. For practical applications, there are many more things to be considered. Some examples are:

Transfers When travelling via public transport one has to consider transfers between different stations. Walking between stations might be beneficial for the travel-time (or even be neccessary to reach the target at all). But how do you find reasonable station-pairs between which the user should be able to walk. Considering all pairs of stations would be too costly to compute regularly. Most scientific reports use a subset of those pairs, but only rarely give further information about the exact set or how it was computed.

The easiest approach would be to use a threshold-value, e.g. let the user walk between stations if they are closer than MAX_WALK m. However this choice is rather arbitrary.

Changing Time The consideration of changing times (to change from one vehicle to another) is important for real-world application. Again we could derive values ad-hoc via a constant. But depending on the modelling of the network this might not be a good idea. Considering a large train-station: Changing between adjacent platforms is quick. Going from one end of the station to the other end can take much longer though.

Source stations When one starts a journey from his own home, we mostly don't mind whether we start from the bus station 2 minutes down the road or from the train station which is a 5 minute walk away. To capture this we would need to start our queries not from a single source stop, but rather a set of source stops. Similarly we most likely have several target stops as well.

Service days Most Transportation Networks offer different schedules on different days (e.g. work-days, weekends, holidays, etc.). These schedules might or might not repeat weekly. This means one either needs to update the data-structures regularly (daily?). Or encode all possibilities into the data-structures and filter unavailable connections when the query is executed.

Especially for algorithms that use preprocessing the regular recomputation could be a considerable time investment.

2.4 Map Matching (Application of RP techniques)

Nowadays, almost every cell phone is equipped with GPS or at least allows for localization derived from nearby cell phone base stations. This gives rise to a plethora of exciting applications as well as research questions.

From an application point of view, often we are more interested in higher level location information like *Where has the mobile user moved along in the road network?* rather than a pair of coordinates specifying its position in longitude and latitude. The generation of such higher level location information becomes even more challenging when the location measurements are imprecise. GPS (in the non-military context) typically incurs imprecisions in the range of several meters. If GPS is not available, for example due to obstructions by tall buildings in an urban environment or foliage, localization derived from nearby cell phone base stations allows for location measurements with an uncertainty of several hundred meters to few kilometers.

In this subsection we consider the *Map Matching Problem*, that is, given a sequence of (imprecise) location measurements of a mobile user and an underlying road network, compute a reasonable route where the user has traveled along – in other words we aim at matching a discretized and fuzzy trajectory to a route in a road network. We are particularly concerned with the scenario where the location measurements are very imprecise (imprecision up to few kilometers), since there are still numerous situations where more precise location information is not available or can only be acquired at high cost. The proposed solutions makes use of the speed-up techniques developed at the beginning of this chapter.

A *road network* is an edge-weighted directed graph $G = (V, E, c)$ whose vertices are mapped to pairs of coordinates (usually longitude and latitude), and whose edges bear a cost c (typically travel time). A *location measurement* is a triple (x, y, r) consisting of longitude and latitude coordinates x and y, respectively, and a radius r. That is, we model a measurement by a circle and the degree of uncertainty about the true location at the time of measurement, also referred to as *imprecision*, is captured by the radius of this circle.

The input to the *map matching* problem consists of a road network $G = (V, E, c)$ and a sequence $M = m_1, m_2, \ldots, m_s$ of location measurements that have been taken while traveling along a *path* $\pi = v_0, v_1, \ldots, v_k$ in G, that is, a sequence of vertices such that there is an edge directed from v_{i-1} to v_i in G for all $i \in \{1, \ldots, k\}$. From M we want to reconstruct π, that is, find a path $\pi' = u_0, u_1, \ldots, u_{k'}$ in G that is close to π, see Figure 4 for a simple problem instance.

2.4.1 Standard Approaches

- point-to-point matching
- naive layered graph approach

2.4.2 Refined Layered Graph Approach

- single source-target query with dummy source

2.4.3 Speeding-Up via CH

- many one-to-one queries
- reuse up-search
- tagging

2.4.4 Further Challenges

Determining 'nearby' Nodes

- Range-Tree

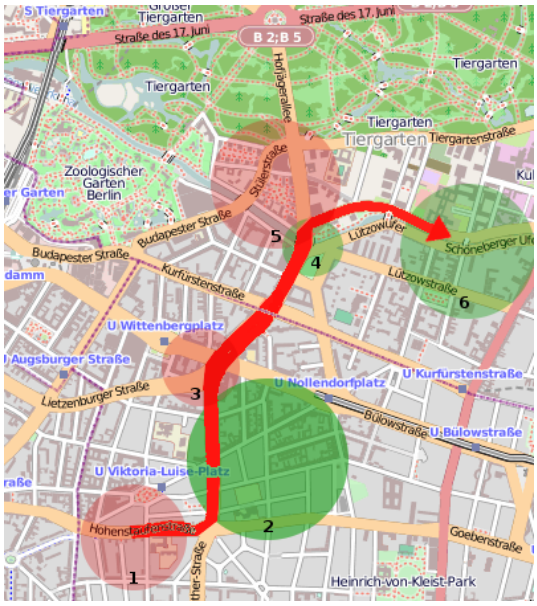


Figure 4: The Map Matching Problem: We are given a sequence of (here: 6) imprecise location measurements and want to reconstruct the original (red) path

- kd-Tree
- Grid

2.5 Energy-efficient shortest Paths (CH extension)

See [14] (freely available online).

3 Information Retrieval

Given a collection of text documents (e.g., web pages), we are to determine all documents which contain all keywords of a given query. If the total length of the text documents is n and the total length of the keywords is k, a naive solution can be achieved in $O(nk)$ by scanning through all documents and trying at each position whether a keyword is present there. The Rabin-Karp algorithm with rolling hash functions or the Knuth-Morris-Pratt algorithm can improve this to $O(n+k)$ in practice or even theory, yet $\Omega(n)$ time is still unacceptable for really large text corpora (like the internet). While a GB of data can be scanned within a fraction of a second, large text corpora have sizes in the Tera- or even Petabytes range (the 'web' contains an estimated 2500 TB of text as of 2016). In the following we will briefly present solutions which after a preprocessing phase of the text documents allow for much faster query times.

3.1 Inverted Index

Assume the text corpus consists of distinct text document each with a unique ID. The idea of the inverted index is very simple: for each word we construct a (sorted) list of IDs of documents containing that specific word. Queries can now be answered as follows:

- if our query consists of a single word, looking up the respected list of IDs.
- if our query consists of two words which both should appear in the documents, we look up the two lists L_1 and L_2 and compute their intersection in $O(|L_1| + |L_2|)$.
- if our query consists of two words of which at least one should appear in the output, we look up the two lists L_1 and L_2 and compute their (sorted) union in $O(|L_1| + |L_2|)$.
- if our query consists of more words all of which should appear in the documents, we can perform a sequence of pairwise intersects or a k-way merge ...

Actual Construction of an Inverted Index We create a **map** from strings to arrays of ints, iterate over all documents, and in each document iterate over all words and add the respective document ID to the according map entry. Looking at the frequencies how often words appear in the documents, we can often observe them adhereing to *Zipf's law*: the i-th most frequent word appears proportionally to $1/i^\alpha$ for some constant α.

Efficient List Intersections Let us consider the problem of intersecting two lists A and B containing document IDs in more detail. Assume $|A| = k < n = |B|$ and both lists are sorted. The straightforward solution is a simple merge in $O(k + n)$ which seems to make sense if $k \approx n$. On the other hand, if $k << n$, binary search might be beneficial: we search each element of A in B and end up with a running time of $O(k \log n)$.

There is still some potential for improvement, though. assume the first element of A locates itself very much at the very end of $|B|$, then searching the remaining elements of A in B at cost $O(\log n)$ each seems wasteful. In fact, if we have located an element of A in B and the next element of A is 'near' in B, less than $\Theta(\log n)$ effort should be possible. And indeed, this is possible. IF $A[i]$ and $A[i+1]$ are at positions j_1 and j_2 in B, then we can locate $A[i+1]$ in time $\log d$, where $d = j_2 - j_1$ by first performing an exponential search from j_1 to determine the rough range in B where to search for and then performing binary search on this range. This is called finger or gallopping search.

In general, let $j_1, \ldots, j_k$ be the positions of the elements of A in B, $d_i = j_i - j_{i-1}$ for $i > 1$, and $d_1 = 1$. Then the time complexity for list intersection is $O(\sum \log d_i)$. Under the condition that $\sum d_i \leq n$, this sum achieves its maximum when $d_i = n/k$, namely $O(k \log n/k)$. This is always $O(n)$ and hence better than the naive solution, even if $k \approx n$. Note that this is also optimal in a comparison-based model, since there are $\binom{n+k}{k}$ ways how A 'merges' into B, hence we need at least $\log \binom{n+k}{k} \geq \log(n/k)^k = k \log(n/k)$ comparisons.

Further remarks:

- modern processors make heavy use of pipelining and speculative execution of future instructions; for this to work efficiently, conditionals should be avoided if possible or made as predictable as possible; example: use sentinels to avoid testing for index out of bounds; in our case: add ID ∞ at the end of each list.

Caveats For this to work well in practice, one has to regularize the words in the documents in some way:

- 'school' should be treated the same as 'School'
- 'bus' should be treated the same as 'buses'
- 'Käfer' ... 'Kaefer'?
- Donaudampfschifffahrtskapitän ... Donau, Dampf, Schiff, Fahrt, Kapitän?

Ranking and Evaluation Very often, a query returns a huge number of hits (think of all movies containing the words 'vampire' and 'zombie'). Hence for a search engine to be useful, the *ranking* of the results is of highest importance. Relevant results should appear early on in the list of search results. We describe the basic problem and a simple solution in the following:

Let us assume we have an augmented inverted list in which for a term, each document also has a *relevance score* associated. When merging/intersecting result lists these scores could be *aggregated* (e.g., summing the relevance scores). In its simplest case, the relevance score is always 1 and we order the documents according to how many of the desired keywords appear in them (not counting multiplicities). For example, if searching for 'zombie OR vampire', movie descriptions containing both 'zombie' and 'vampire' will get a higher score (2).

With several hundreds, thousands, or even millions of results, one is typically only interest in a few of the top ranked results. While sorting all results takes $O(n \log n)$ for a result list of length n, Extracting the top k results from a heap costs $O(n + k \log n)$, which for small k is $O(n)$.

Assigning a meaningful relevance score is a problem on its own. A straightforward interpretation is the frequency how often a term appears in the document. There are some problems with using this as a relevance score. First, long documents typically have larger such value, obviously. Here ist might be beneficial to normalize by the maximum number of occurrences of any term in the document – this is called *term frequency* and can be defined as $tf(t,d) = \frac{f(t,d)}{\sum f(t',d)}$ (similar other definitions exist).

There is another problem, though, that some words like 'of', 'the', 'simple', ... appear in almost any document quite frequently. To fix that we first consider for every term in how many documents it appears – this is called the *document frequency (df)*. Then we relate this to the total number of documents N: $\log N/df$ also taking the log to obtain the *inverse document frequency (idf)*. In the actual ranking, idf and tf are then often multiplied (more complicated ways, also taking into account the length of the documents, are used in practice; *BM25 formula*). Also, it might be interesting to determine a quality metric for the documents itself (independent of the terms appearing in it); the PageRank can be interpreted as one incarnation of that.

Compression In particular for very common words, the number of document IDs per term might become very large, just scanning these lists is expensive, in particular if they reside on harddrive (transfer rate of main memory $\approx$ 2GB/s, of HDD 50GB/s). Depending on the compression rate and the time to decompress, compression often pays off. A simple way is to use *gap encoding* which is particularly suitable if the document IDs are stored increasing order. Instead of storing the full document ID, only the difference to the previous document ID is stored. Typically these are relatively small numbers (compared to the document IDs itself) and hence should be representable in less space. Note that naively encoding these deltas as binary numbers as $0 = 0$, $1 = 1$, $2 = 10$, $3 = 11$, ... does not work, as this decoding is not unambiguous. So one has to use a prefix-free

code, like Elias-Gamma ($\lfloor \log_2 x \rfloor$ zeros, then x, which has size exactly $2\lfloor \log_2 x \rfloor + 2$) bits) or even better a Huffman code. All these schemes use somewhat nasty bit operations, often they are used together with *variable byte encoding* where we always use full bytes for the code with the first bit of each byte indicating whether this is the end of the code word.

3.2 Substring Search via Suffix Trees and Suffix Arrays

So far we have only considered retrieval of complete words in a given set of text documents. Sometimes it is useful, though, to allow for arbitrary substring searches. Using the same approaches like an inverted index would blow up the number of 'words' considerably, though, so other approaches are needed.

Formally, we are given a large text T of length $|T| = n$ and we want to preprocess T such that subsequently we can determine for any pattern P with $|P| = m$ all its occurrences in T. Here we think of $m << n$ and hence aim to avoid a running time linear in n.

3.2.1 Suffix Tree

First we assume that T ends with a special character \$ which assures that no suffix is a prefix of another suffix. A suffix tree for T is a tree with the following properties:

- every suffix of T corresponds to a leaf of the suffix tree
- each edge in the tree is labeled with a non-empty string
- the edge labels to the i-th leaf of the tree yield the i-th suffix
- every internal node has at least two children

Example: banane.

While this tree structure has linear size, the string labels on the edges might consume a lot of space; but we can replace them by simple indices into the text T.

Naively, the tree can be constructed incrementally as follows (here T_i denotes the suffix starting at the i-th letter of T:

1. to insert T_i we traverse the existing tree according to the letters in T_i until a mismatch occurs:
   - if the mismatch occurs in the 'middle' of an edge, split the edge and branch off the newly inserted vertex
   - otherwise branch off the the respective vertex

   Example: banane

Inserting all n suffixes sums up to a total cost of $O(n^2)$; but there are much more efficient ways of construction, in particular a linear time algorithm for constructing suffix trees by Ukkonen [15].

Searching We observe that every substring is the prefix of some suffix of the text. We simply traverse the suffix tree according to the given pattern P. If we get stuck, there is no match in T. If we end in a node u, set $x = u$, if we end in the middle of an edge (u, v), set $x = v$. Then all leaves below x represent matches in T.

Example: banane

Using this scheme we can find all j occurrences of a pattern P in time $O(|P| + j)$ (assuming constant-sized alphabet).

3.2.2 Suffix Arrays

Suffix arrays are a data structure serving the same purpose but being more space efficient (better constants, even though suffix trees also have size $O(n)$). As a drawback, they exhibit a query time of $O(|P| + j + \log n)$. The suffix array is essentially just the lexicographic order of all suffixes in the text. Naive construction can be performed in $O(n^2 \log n)$, alternatively one can first construct a suffix tree and then convert this into a suffix array. As we do not have to store the suffixes explicitly (indices suffice), suffix arrays are very space efficient.

Search The search for a pattern is simply a binary search which costs $O(m \log n)$ since every comparison might have to look at m characters. This can be further tuned to $O(m + j + \log n)$ via a *longest common prefix array* where we store for each T_i the length of the common prefix with T_{i-1}. In the binary search procedure we consider a left element l, a right element r and a middle element m. We compare P with the element at m and recurse into $[l, m]$ or $[m, r]$ or determine that P equals the suffix at position m. Let k be the length of the longest common prefix of P and the m-th suffix and assume that P is lexicographically smaller than the m-th suffix. Furthermore let m' be the middle element in the next recursion on interval $[l, m]$, and K the rlcp length of the suffix starting at m. Distinguish three cases:

- $k < k'$: the m'-th suffix and the m-th suffix share the first $k+1$ characters, hence we continue again in the left half without any character comparison
- $k > k'$: the m'-th suffix is lexicographically smaller than the m-th suffix (due to its position); hence the $k' + 1$-th character of the m'-th suffix is smaller than that of the m-th suffix (and hence of P!); we continue in the right half without any character comparison
- $k = k'$: we compare by starting a comparison at character $k + 1$ and proceed accordingly

In a nutshell, this procedure only needs to look at a character of P once, hence the running time is $O(\log n + |P|)$. Note that we assume here that we can determine in $O(1)$ the length of the lcp of the m-th and m'-th suffix which are not next to each other. It is not difficult to see, though, that this is in fact the minimum of the precomputed lcp lengths for $m' + 1$, $m' + 2, \ldots m$.

Efficient Construction One can turn a suffix tree into a suffix array in $O(n)$ time, hence with a $O(n)$ suffix tree construction, there is also a $O(n)$ suffix array construction. We present some simple, direct methods which improve upon the naive $O(n^2 \log n)$ construction.

$O(n \log^2 n)$ **construction:** The idea is to first sort the suffixes by the 1st character only, then by the first 2 characters, then by the first 4 characters, etc. Note that when we have sorted according to the first 2^i characters, and we want to sort according to the first 2^{i+1} characters, we can do so in $O(n \log n)$ time since the comparison between two suffixes takes $O(1)$.

A linear time algorithm for construction was presented in [16], in particular pages 5–7.

Digression: Integer Sorting

During the linear-time construction of suffix arrays we assumed that integer numbers could be sorted in linear time. In the following we want to give a brief account on under what circumstances this is possible and why this does not contradict the $\Omega(n \log n)$ lower bound for comparison-based sorting.

Assume we are to sort n integer numbers, each consisting of d digits, furthermore assume d being constant (e.g., in a typical computer, we have words of 64bit length, hence $d = 64$). The following *radix sort* algorithm sorts these numbers in time $O(nd)$ which is $O(n)$ if d is treated as constant. Given a sequence of the n integers to be sorted, the first pass iterates over all integers from the beginning, rearranging in a stable manner the sequence such that all integers with *least significant* bit 0 are followed by all integers with least significant bit 1. 'Stable' here means that the order of two integers with equal least significant bit does not change from the original to the rearranged sequence. Obviously, this can be done in $O(n)$ time. The same process is repeated for

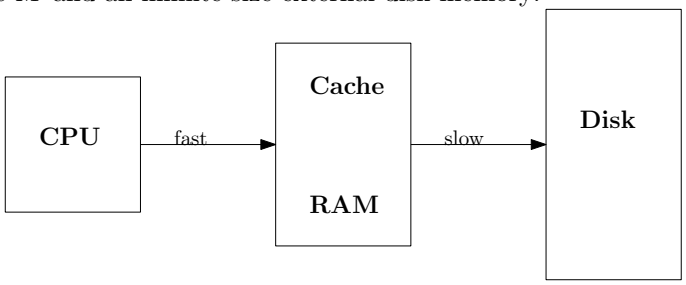
the 2nd least significant bit, the 3rd least significant bit until al d bits have been considered. The overall running time is $O(nd)$ and the sorting is correct due to the following reasoning: consider two numbers a and b, $a \neq b$; let i be such that the i-th bit is the most significat bit where a and b differ. Clearly the order between a and b during radix sort does not change while considering even more significant bits. The last iteration where a and b were swapped is the iteration where the i-th bit is considered. But here the smaller of the two numbers is put before the other.

Radix sort (and other 'linear time' sorting algorithms) of course are only linear time if d is treated as being constant.

4 Memory Hierarchies

If the data to be processed does not fit into main memory, actual running times of actual algorithms are mostly determined by the number of I/O operations that read and write data from/to external memory. In reality, there are many different 'memories': CPU registers, L1 cache, L2 cache, L3 cache, RAM, hard disk, which increase in size (from few Bytes to tera- or petabytes) but also access time (ns to ms) and bandwidth.

So while typically algorithms are designed with minimization of the number of CPU operations in mind, in the 'big data' setting, they should be tuned to minimize I/O operations. For simplicity we assume the following established external memory model where we have a fast cache/RAM of size M and an infinite size external disk memory:



We further assume that both Cache/RAM as well as the disk is divided into blocks of size B, so the cache holds M/B blocks. Transfering one block from cache to disk or the other way around has assumed cost 1. Clearly, an algorithm which for every data access requires disk access might cause as many expensive disk accesses as its running time $T(N)$. On the other hand, a carefully I/O-tuned algorithm might group data accesses such that only $T(N)/B$ disk accesses happen.

In the following we will investigate some basic algorithmic problems and how they can be addressed in the I/O model.

4.1 Basic Problems

4.1.1 Scanning and Selection

Obviously, linearly scanning an array of N items costs $O(N/B)$ I/O operations which is also optimal. But let us consider a slightly more involved problem for which we also know a $O(N)$ time algorithm: median computation or selection. The following 'median-of-5s' algorithm selects the rank-k element of an array $A[1...N]$ in $O(N)$ time in RAM:

```
select(A,k)
 group A into groups of 5
 compute the median in each group
 recursively compute the median m5 of the medians
 partition A according to m5
 recurse into the half containing the rank-$k$ element
```

The key insight for the standard analysis is that m_5 has essentially $N/10$ other 5-medians to both sides and hence also $3N/10$ of all elements, so it is a somewhat close to the true median. The running time resolves hence to $T(N) = O(N) + T(2N/10) + O(N) + T(7N/10) = O(N)$.

Fortunately, all operations of this algorithm consist of simple scans over A and sequential write operations, hence we obtain an I/O complexity of $O(N/B)$ which is also optimal (think of just the number of I/Os to read A from disk once).

4.1.2 Search Trees

Assume we are to manage a totally ordered set of N elements that is too large to fit into main memory. A straightforward strategy is to construct a B-tree with branching factor $B + 1$, that is, every internal node holds B keys. The resulting tree has depth $O(\log_{B+1} N)$. All common

operations on the tree like insert, delete, and lookup can then be performed in $O(\log_{B+1} N)$ I/O. This is also optimal as one can see as follows for the search operation: locate a query element q within the N elements $\log N$ bits are necessary. Reading a block with B keys reveals where q locates within these B elements, that is, $\leq \log B$ bits of information. Hence we need at least $\frac{\log N}{\log B} = O(\log_B N)$ I/Os.

Exercise: What is the I/O complexity of binary search?

4.1.3 Sorting

In the RAM model, B-trees also yield an asymptotically optimal sorting algorithm by simply inserting all elements one-by-one. We could also do this in the external memory model to obtain an algorithm with $O(N \log_B N)$ I/Os, which is *not* optimal. Modifying mergesort, though, yields an optimal external memory sorting algorithm:

Divide N elements into N/M groups of size M each. They can be sorted internally, hence we get N/M sorted runs. We merge those runs in k-way merges with $k = M/B$ (which costs $O(N/B)$ I/Os per level). Hence the recursion tree has depth $O(\log_{M/B}(N/M))$ and overall I/O complexity is $O(N/B \log_{M/B} N/M)$..

4.2 Cache Obliviousness

So far we have assumed that we know about the sizes of M and B. Yet, in practice, these may differ from platform to platform, and if there are more than 2 levels in the memory hierarchy, it would be nice if an algorithm 'automatically' adapts to the memory hierarchy. Of course, any ordinary algorithm is in some sense cache oblivious, but the question is how well does it make use of the cache(s), or, how well-structured are the accesses across the memory hierarchy. In terms of an implementation, cache-obliviousness also is much easier than explicitly managing blocks or pages on the hard drive. When some data item is accessed, the whole block containing the item is read, and the lest recently used (LRU) block is evicted from cache if necessary. While the paging strategy LRU is not necessarily optimal, one can prove 2-competetiveness compared to the optimum strategy with half the cache size (just partition block accesses into phases of M/B distince blocks).

4.2.1 Scanning and Selection

Are inherently cache oblivious already.

4.2.2 Search Trees

The B-tree layout described above inherently uses knowledge about B, the following *van Emde Boas* layout can be used to yield cache-oblivious search trees.

We first store all N elements in a complete binary search tree and cut that tree at half-height $(0.5 \log_2 N)$. The upper half is then a tree containing $\sqrt{n}$ elements, the lower half contains $\sqrt{n}$ trees of size $\sqrt{n}$ each (and height $0.5 \log_2 N$). We recursively lay out these trees on disk. Let us now consider a search in this tree layout and the level of recursion where the next recursion level contains all elements of a subtree in one block, that is, it has between $\sqrt{B}$ and B elements, or, height between $0.5 \log_2 B$ and $\log_2 B$. Any leaf-to-root path visits at most $\log_2 N/(0.5 \log_2 B) = 2 \log_B N$ such subtrees. Each of these subtrees occupies at most 2 blocks on disk, hence overall we need $O(4 \log_B N)$ I/Os for a search.

This can also be extended to update operations on the tree (no details given here).

4.2.3 Sorting

Exercise: What is the I/O complexity of cache oblivious standard binary mergesort?

Under the 'tall-cache assumption', $M = \Omega(B^{1+\epsilon})$, e.g., $M = B^2$ or $M/B = \Omega(B)$, there is a cache oblivious variant of merge sort ('funnel sort'); no details given here.

5 Beyond Worst Case Analysis

In many cases, worst case analysis is overly pessimistic and also fails to explain the performance of algorithms and data structures in practice. We will consider two alternatives to worst case analysis, one is more geared towards data structures, the other more towards algorithms.

5.1 Amortized Analysis

For data structures, worst case analysis is often overly pessimistic. Let us consider as an example the very simple 'data structure' of a counter consisting of k bits which can represent the number 0 to $2^k - 1$. The only operation of this data structure is the increment operation. The cost of such an increment operation is the number of carry-overs which can be as much as almost k, e.g., consider the increment operation on the number $2^{k-1} - 1 = 01111\ldots1$. So in worst case terms, the cost of a single increment can be k. But now consider a sequence of increments. Can it happen that each of these increments is so costly, always creating $\theta(k)$ carry-overs?

It turns out, that this is not the case, and amortized analysis is one way to see that. We first define a so-called *potential* $\phi : \mathcal{S} \to \mathbb{N}$ which assigns each state from $\mathcal{S}$ of a data structure (in our example: the value currently represented) some number. Now if we have an operation op on the data structure which brings the data stucture from state $s_1 \in \mathcal{S}$ to state $s_2 \in \mathcal{S}$ and incurs real cost $rcost(op)$, we define its *amortized cost* as $acost(op) = cost(op) - \phi(s_1) + \phi(s_2)$.

Now consider a sequence of operations $op_1, op_2, \ldots op_l$ which have real cost $\sum_{i=1}^{l} rcost(op_i)$. The amortized cost of this sequence of operations is

$$\sum_{i=1}^{l} acost(op_i) = \sum_{i=1}^{l} rcost(op_i) - \phi(s_1) + \phi(s_{l+1}) = \phi(s_{l+1}) - \phi(s_1) + \sum_{i=1}^{l} rcost(op_i)$$

Here the last equality is due to all other ϕ-terms cancelling out. So the sum of the amortized costs of all operations are essentially the sum of the real costs of the operations with two terms added which can be neglected if they are not too big compared to the sum of the real costs.

Now if we could show somehow that the amortized cost of an individual operation is $O(1)$ we could hence conclude that the sum of amortized and hence also the sum of real costs of all operations is $O(l)$!

In our example, this is quite easy. Simply choose as potential function ϕ the number of 1's in the binary representation of the current value represented. The real cost of an increment is 1+ the number of carry-overs; but with every carry-over one 1 disappears from the binary representation of the current value represented, hence the potential drops by one. Therefore the real cost is in fact balanced out by the change in potential and the amortized cost of an increment turns out to be $O(1)$! We conclude, that for a sequence of l increments, the real cost is

$$\sum_{i=1}^{l} rcost(op_i) = \phi(s_1) - \phi(s_{l+1}) + \sum_{i=1}^{l} acost(op_i) = O(l)$$

In class we have also seen a more involved example for $(2, 4)$-trees, here we could prove that in a sequence of insertions and deletions, the effort for reorganising the tree (via split or merge operations which in the worst-case can be $O(\log n)$ due to the height of the tree) have $O(1)$ amortized cost.

5.2 Smoothed Analysis

Another alternative to worst-case analysis is the so-called *average case analysis*. Here, the expected performance (which might be running time or quality) of an algorithm is analyzed for inputs that are generated according to a known probability distribution. One example could be, quality for some heuristic for the TSP problem, where the cities are drawn uniformly at random from the unit

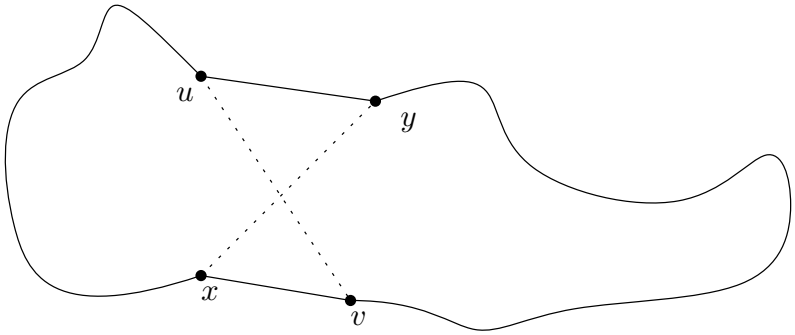


Figure 5: Swap operation in 2OPT.

.

square. Or the running time of some local search algorithm on an instance of the TSP problem that was also drawn uniformly.

Yet this kind of analysis is not really capable of explaining real-world performance of algorithms since real-world very rarely are generated according to such a simple probability distribution. In fact, most of the time it is very difficult to compactly express any probability distribution to which real-world instances adhere to.

The so-called *smoothed analysis* is in some sense something inbetween worst-case analysis and average case analysis. Here the idea is that we are allowed to consider an arbitrary problem instance (like in worst case analysis), but then perturb some parameter of the instance by some amount σ. Then the goal is to prove a bound on the performance (dependent on σ). For σ very large the result can be viewed as similar to an average case analysis, vor σ very small, we are at worst-case analysis. We will give one simple example for this proof technique in the following.

5.2.1 2-OPT Local Search for TSP

One very popular heuristic (without good approximation guarantee, though) for TSP is a local search technique called *2-Opt* which works as follows. Given some solution (tour) to a TSP instance (of arbitrary quality), one inspects all pairs of edges $\{u, v\}$ and $\{x, y\}$ that are part of the tour and checks whether swapping these edges to $\{u, y\}$ and $\{x, v\}$ improves the cost, see Figure 5 for an example. In each 'round' at most $O(n^2)$ edge pairs have to be considered until a swap (and hence improvement of the tour quality) happens, or no swap improves the cost and hence the 2OPT-algorithm has finished. Unfortunately, no good bound on the quality of the solution is known (there are instances where the outcome is $\Omega(\frac{\log n}{\log \log n})$ more expensive than the optimum), yet in practice the solutions are often close to optimal. In the following we want to analyze how many rounds this 2OPT-algorithm might perform. There are worst case instances, where an exponential (in n, the number of vertices/cities) number of swaps occurs, yet in practice considerably less steps are typically taken until no further improvement is possible. Smoothed analysis is one attempt of an explanation. For sake of a simpler exposition we assume the L_1 metric, that is the distance between x and y is simply $|x_1 - y_1| + |x_2 - y_2|$.

So let us make the model for the analysis a bit more precise. We assume an *arbitrary* set of points $p_1, p_2, \ldots p_n$ given in the unit square $[0, 1]^2$ and add to each point some noise (e.g., random perturbation within a square of side length σ around the original point). This model can also be expressed as having for each point p_i a density function f_i with $f_i(x) \leq 1/\sigma$ (this is in fact even more general than random perturbation within a radius of σ).

Now consider one concrete swap as in Figure 5 and assess the change in the tour cost when replacing edges $\{u, v\}$ and $\{x, y\}$ by $\{u, y\}$ and $\{x, v\}$. We have:

$$\Delta = ||u - y|| + ||x - v|| - ||u - v|| - ||x - y||$$

$$= |u_1 - y_1| + |u_2 - y_2| + |x_1 - v_1| + |x_2 - v_2| - |u_1 - v_1| - |u_2 - v_2| - |x_1 - y_1| - |x_2 - y_2|$$

A swap takes place if $\Delta < 0$. Let us call a swap ϵ-*bad*, if $\Delta > -\epsilon$ for some small ϵ.

Lemma 4. *For every perturbed instance and $\epsilon > 0$, the probability that there exists a ϵ-bad swap is $O(\epsilon n^4/\sigma)$*

Before we prove this Lemma, let us state and prove the final Theorem using this Lemma.

Theorem 1. *The smoothed complexity of 2OPT for TSP under the l_1 metric is $O(\sigma^{-1} n^6 \log n)$.*

Proof. Any tour has length at most $2n$. If there are no ϵ-bad swaps, 2OPT must terminate in $2n/\epsilon$ rounds. Therefore (since there are at most $n!$ rounds):

$$E(\#rounds) = \sum_{M=1}^{n!} Pr(\#rounds \geq M) \leq \sum_{M=1}^{n!} Pr(\exists \frac{2n}{M} - \text{bad swap}) \leq \sum_{M=1}^{n!} O(\frac{n^5}{M\sigma})$$

$$= O(\frac{n^5}{\sigma} \sum_{M=1}^{n!} 1/M = O(\frac{n^6 \log n}{\sigma}))$$

□ □

And for the proof of the Lemma:

Proof. There are only $O(n^4)$ different swap configurations, so let us fix one of these and try to bound the probability that it is ϵ-bad. For the involved 4 points there are $(4!)^2$ different relative orderings (considering the two dimensions separately). Fixing one relative ordering, the expression for Δ can be written down without the absolute value operator and becomes a linear term with 8 variables with coefficients all in $\{-2, 0, 2\}$. Δ is in $(-\epsilon, 0)$ only if at least one of these linear combinations lies in $(epsilon, 0)$. Consider one of these linear coefficients and a variable with non-zero coefficient, w.l.o.g. u_1 with coefficient 2. We then have $\Delta = z + 2u_1$. Having fixed all other variables already, we ask for what range of u_1 is Δ in $(-\epsilon, 0)$. This range has size at most $\epsilon/2$, hence $Pr(u_1 \text{lies in interval of size} \epsilon/2) \leq \epsilon/(2\sigma)$ (due to the property of the density function). Via the union bound over all possible swap configurations we obtain that the probability for the existence of an ϵ-bad swap is $O(n^4\epsilon/\sigma)$. □

5.2.2 Knapsack

In class we also performed a smoothed analysis of the expected number of pareto-optimal solutions of a smoothed knapsack instance. This is also not relevant for the examination.

$$
\begin{array}{lrcrcrcrcr}
\min & 7x_m & + & 3x_t & + & 2x_b & + & 4x_c & & \\
\text{s.t.} & 1x_m & + & 2x_t & + & 4x_b & + & 1x_c & \geq & 11 \\
 & 3x_m & + & 2x_t & + & 1x_b & + & 4x_c & \geq & 7 \\
 & 5x_m & + & 0x_t & + & 0x_b & + & 2x_c & \geq & 5 \\
 & x_i \geq 0 & & & & & & & &
\end{array}
$$

Figure 6: Example for an (I)LP.

6 (I)LP-based Techniques

(Integer) Linear Programming has become a very important optimization technique with countless application domains. Many problems in business or industry can be formulated as (integer) linear programs and hence a lot of manpower has been spent on (I)LP solvers that can deal even with very large problem instances. Figure 6 shows a simple linear program, consisting of an objective function (in this case to be minimized) and several constraints, all of which are linear in the variables x_m, x_t, x_b, x_c – hence the name *linear program*. Depending on the domain of the variables, we have a 'normal' *linear program (LP)* (if $x_m, x_t, x_b, x_c \in \mathbb{R}$) or an *integer linear program (ILP) (if* $x_m, x_t, x_b, x_c \in \mathbb{Z}$*)*.

From a theoretical point of view, the case with continuous variable domains (LP) is 'easy' in that there are polynomial-time algorithms (in the number of variables and constraints) computing an optimal solution, e.g. the *Ellipsoid method* [**?**] or *interior point* methods. The case with discrete variable domains (ILP) is considerably harder, as computing an optimal solution to a general ILP is easily shown to be NP-hard. For both cases (LP and ILP), an optimal solution can be found in time *linear* in the number of constraints if the dimension is considered fixed.

In practice, LPs with thousands of variables and constraints can often be solved within few seconds, end even ILPs stemming from real-world problem instances can often be solved up to moderate size of several hundreds of constraints or variables. Commercially developed solvers like GUROBI or CPLEX have reached an astonishing level of maturity but are also quite expensive. For many problems, open source projects like GLPK or LPSolve also suffice, though.

For more linear programming background we refer to the lecture *discrete optimization*.

In the following we will sketch two applications of (integer) linear programm techniques that have proven to work very well in practice (in particular much better than theoretical running time guarantees suggest).

6.1 Multicriteria Contraction Hierarchies

While conventional route planning engines usually compute the shortest or quickest path between a given source and a target, individual preferences of users might differ. For example, a user might accept a slightly later arrival time at the target in exchange for a less crowded route, or reduced gas consumption, or fewer traffic lights or left turns on the way. To provide full flexibility, all possible trade-offs between all criteria should be valid definitions of an optimal route – and each query should be allowed its own definition. The personalized route planning problem captures this idea. Formally, it can be phrased as follows: Given a street network $G(V, E)$, with a d-dimensional non-negative cost vector $c(e) \in \mathbb{R}^d$ for each edge $e \in E$ (where each entry reflects one criterion, e.g. c_1 travel time, c_2 number of traffic lights, c_3 gas price, and so on); a query consists of source and target $s, t \in V$ and non-negative weights $\alpha_1, \alpha_2, \cdots, \alpha_d$ (expressing the importance of each cost component for the user). The goal is to compute the path p from s to t in G which minimizes $\sum_{e \in p} \alpha^T c(e)$.

Clearly, the personalized route planning problem can easily solved by Dijkstra's algorithm where before each edge relaxation its weighted scalar cost ist determined using the weights α_i. But again, as for the single-metric case, we are interested in speed-up schemes that – after a preprocessing phase – allow for faster queries. In the following we will sketch an adaptation of the previously covered *contraction hierarchies* for the multi-metric case.

As for the ordinary CH construction, the central operation is the *node contraction*. Here, when contracting a node v one has to decide wheter to put a shortcut (u, w) between neighbors u, w of v with $(u, v), (v, w) \in E$. In the original setting, the decision was simple: if uvw is the only shortest path, the shortcut (u, w) has to be created. In the multicriteria setting, we have to create the shortcut (u, w), if there exist weights α_i for which uvw is the only optimal path. The challenge here is that there are infinitely many weights α_i which are hard to check. Fortunately, linear programming comes as a rescue here.

Let $c_\pi(\alpha)$ be the cost for path π under weights $\alpha = (\alpha_1, \ldots, \alpha_d)$ (which is a linear expression in the α_i). We consider the following linear program in variables $\alpha_1, \ldots, \alpha_d$ and ϵ:

$$\begin{array}{lllllll} \max & & \epsilon & & & & \\ \text{s.t.} & \forall \text{u-w-paths } \pi' : & c_{uvw}(\alpha) & + & \epsilon & \leq & c_{\pi'}(\alpha) \\ & & & & & \alpha_i \geq 0 & \\ & & & & & \epsilon \geq 0 & \end{array}$$

If this LP has an optimum solution with $\epsilon > 0$ we know that there exist values $\alpha_1, \ldots, \alpha_d$ such that for this choice of α_i the path uvw is at the only optimal u-w-path. The drawback of this formulation is, of course, that we need a constraint for *every* possible u-w-path – typically there are exponentially many of them. Fortunately, we can use the following approach:

1. set up the LP with only the $\alpha_i \geq 0$ and $\epsilon \geq 0$ constraints
2. solve the LP; if it is infeasible, dismiss the shortcut and abort, otherwise retrieve the respective α_i values
3. compute the shortest path π' from u to w under the retrieved α_i values; if it is uvw, create shortcut and abort
4. add the constraint for π' and go to step 2.

Clearly, this procedure terminates in finite time (as there are only finitely many u-w-paths) and certifies either that the shortcut is necessary or that it can be dismissed. Step 3. can easily implemented by Dijkstra's algorithm. Interestingly, if the Ellipsoid method is used to solve the LP, one can even show that the optimum solution (or infeasibility) can be certified in polynomially many steps. In practice, employing a dual simplex algorithm also works well as it can make use of the solution computed in the previous iteration. Typically, only very few (around 10) iterations are necessary for a conclusive decision about the shortcut and the overall running time is dominated by the Dijkstra runs, not by solving the LP.

For more details see [17].

6.2 The Travelling Salesman Problem

While the previous problem a polynomial-time solution can be proven, problems that are formulated as integer linear program (ILP) are often NP-hard, so one might expect no solution within an acceptable timeframe. Yet, real-world problem instances are often far from the somewhat contrieved instances that are instrumented for proving NP-hardness. In the following we consider an ILP formulation for the NP-hard (metric) *travelling salesman problem (TSP)*, which is the following: Given n points $v_0, v_2, \ldots, v_{n-1}$ in a metric space with a distance function $d(.,.)$, we are interested in finding a permutation π of the n points such that

$$\sum x_{ij} d(v_i, v_j)$$

is minimized. Obviously, the naive solution of enumerating all possible $n!$ permutations is only viable for miniscule problem instances.

We solve TSP via an ILP formulation where for every edge (v, w) there is a variable x_{vw}, which is 1 if it is part of the tour and 0 otherwise. Clearly, we want to minimize the cost of edges that are part of the tour. Still, there are some constraints necessary to ensure that the selected edges

form a tour at all. So first we demand that for every node v there is exactly one incoming edge and exactly one outgoing edge selected. This might result in several cycles instead of one closed tour, so additionally, we demand that for *any* proper subset of the nodes, there must be at least one outgoing edge from and at least one incoming edge to the set (such a constraint is also called 'subtour elimination constraint'):

$$\begin{array}{lll} \min & \sum x_{vw} d(v,w) & \\ \text{s.t.} & \forall v \in V: \quad \sum x_{v.} & = \quad 1 \\ & \forall v \in V: \quad \sum x_{.v} & = \quad 1 \\ & \forall V' \subsetneq V: \quad \sum_{\substack{e=(v,w) \\ v \in V' \\ w \notin V'}} x_{vw} & \geq \quad 1 \\ & \forall V' \subsetneq V: \quad \sum_{\substack{e=(v,w) \\ v \notin V' \\ w \in V'}} x_{vw} & \geq \quad 1 \\ & x_{vw} & \in \quad \{0,1\} \end{array}$$

Again we have the problem here, that the number of possible subtour elimination constraints is exponentially large (essentially 2^n due to the number of possible subsets V'). But as before we can invoke an ILP solver without any subtour elimination constraint first. It will give as a result one or more tours. In the former case, we are done, in the latter we treat every tour as subset V', create the respective constraints, and then resolve. In practice, this allows the quick computation of optimal tours in the Euclidean plane for 50 or more nodes, even with standard open-source solvers like glpk – note that the naive approach enumerating 50! node orders is completely useless here.

6.2.1 Heuristics

Even though more sophisticated ILP formulations allow for the computation of optimal TSP tours with several thousands of nodes, computing such solutions takes quite some time. In practice, one often resorts to quick heuristics or approximation algorithms. Standard approximation algorithms are the MST-based $2-$approximation or the matching-based 1.5-approximation (this means that the resulting tours are provably at most a factor 2 or 1.5 respectively more expensive than the optimal tour).

Another very popular heuristic is the so-called 2-opt heuristic: Starting with an arbitrary tour, the 2-opt heuristic tries to improve the cost of the current tour by small changes. More concretely, for a tour $v_0, v_1, \ldots, v_{i-1}, \mathbf{v_i}, \mathbf{v_{i+1}}, v_{i+2}, \ldots, v_{j-1}, \mathbf{v_j}, \mathbf{v_{j+1}}, v_{j+2}, \ldots v_{n-1}, v_0$, it checks whether changing the edges (v_i, v_{i+1}) and (v_j, v_{j+1}) to new edges (v_i, v_j) and (v_{i+1}, v_{j+1}) makes the resulting tour $v_0, v_1, \ldots, v_{i-1}, \mathbf{v_i}, \mathbf{v_j}, v_{j-1}, \ldots, v_{i+2}, \mathbf{v_{i+1}}, \mathbf{v_{j+1}}, v_{j+2}, \ldots v_{n-1}, v_0$ cheaper. If so, the switch is performed. The heuristic continues until no improving switch is possible. Clearly, this heuristic terminates, as no tour is encountered twice (as we improve in each step). Still, there are examples, where an exponential number of steps are performed. In practice, though, many instances only lead to linearly many improvement steps and close-to-optimal results (again even though there are examples, where the resulting tour is considerably more expensive than the optimal tour).

References

[1] Ronald J. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In *ALENEX/ANALC*, pages 100–111. SIAM, 2004.

[2] Andrew V. Goldberg, Haim Kaplan, and Renato Fonseca F. Werneck. Better landmarks within reach. In *WEA*, volume 4525 of *Lecture Notes in Computer Science*, pages 38–51. Springer, 2007.

[3] Ittai Abraham, Amos Fiat, Andrew V. Goldberg, and Renato Fonseca F. Werneck. Highway dimension, shortest paths, and provably efficient algorithms. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 782–793, 2010.

[4] Adrian Kosowski and Laurent Viennot. Beyond highway dimension: Small distance labels using tree skeletons. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '17, pages 1462–1478, Philadelphia, PA, USA, 2017. Society for Industrial and Applied Mathematics.

[5] Stefan Funke and Sabine Storandt. Provable efficiency of contraction hierarchies with randomized preprocessing. In *Proc. 26th International Symposium on Algorithms and Computation ISAAC*, pages 479–490, 2015.

[6] Ittai Abraham, Daniel Delling, Andrew Goldberg, and Renato Werneck. A hub-based labeling algorithm for shortest paths on road networks. Technical report, December 2010.

[7] Ittai Abraham, Daniel Delling, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. Hldb: Location-based services in databases. In *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '12, pages 339–348, New York, NY, USA, 2012. ACM.

[8] Holger Bast, Stefan Funke, and Domagoj Matijević. Transit: ultrafast shortest-path queries with linear-time preprocessing. In *9th DIMACS Implementation Challenge—Shortest Path*, 2006.

[9] Julian Arz, Dennis Luxen, and Peter Sanders. *Transit Node Routing Reconsidered*, pages 55–66. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[10] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Efficient models for timetable information in public transportation systems. *J. Exp. Algorithmics*, 12:2.4:1–2.4:39, June 2008.

[11] Hannah Bast. *Car or Public Transport—Two Worlds*, pages 355–367. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[12] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. *Intriguingly Simple and Fast Transit Routing*, pages 43–54. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[13] Daniel Delling, Thomas Pajor, and Renato Werneck. Round-based public transit routing. Society for Industrial and Applied Mathematics, January 2012.

[14] Jochen Eisner, Stefan Funke, and Sabine Storandt. Optimal route planning for electric vehicles in large networks. In *AAAI*. AAAI Press, 2011.

[15] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[16] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.

[17] Stefan Funke, Sören Laue, and Sabine Storandt. Personal routes with high-dimensional costs and dynamic approximation guarantees. In *Proc. 16th International Symposium on Experimental Algorithms SEA*, 2017.