Objective of the present work

- Optimize Road Network
- Optimize Airline Network
- Optimize Rail Network
- Optimize Computer Network
- Optimize Social Network

Spanners and Emulators

For most graph algorithms, the performance depends on the number of nodes and edges of the input graph. The running time can potentially be reduced by altering the graph, in particular by adding or deleting edges. After altering the graph, we wish that the answer to the question we ask concerning the graph (in our case the distances between nodes) does not change by much. When edges are deleted only, we obtain a subgraph, which, if it preserves distances to a certain extent, is called a spanner.

Since no edges can be removed from graphs with large girth without significantly altering distances, graphs with many edges (dense graphs) and large girth are important worst-case instances for spanner and distance oracle constructions. In extremal combinatorics, determining $mg(n)$ is a research field of its own. For example, for $g = 4$ the question is the following: how many edges can be added to the empty graph on $n$ nodes without closing a triangle? Intuitively, the more edges that were already added, the harder it gets to add another one. For $g = 4$, the complete bipartite graph is asymptotically optimal. For general $g$, the construction of the graph is much more involved; for some $g$ the value of $mg(n)$ is not even known.

Erdős’ girth conjecture predicts that, for an integer $k$, $m_{2k+1}(n) = m_{2k+2}(n) = \Omega(n^{1+1/k})$. The corresponding upper bound is known to be tight and the conjectured lower bound is a theorem for certain values of $k$ (1, 2, 3, and 5);

For multiplicative spanners, the tradeoff between space and stretch is well understood. Not so for spanners with additive stretch. Aingworth found a $(1, 2)$–spanner with $O(n^{3/2})$ edges and Baswana found a $(1, 6)$–spanner with $O(n^{4/3})$ edges. Wood-ruff gives a strong lower bound for additive graph spanners independent of Erdős’ girth conjecture. He proves that for an integer $k = o \lg \lg n$, there are graphs for which any $(1, 2k - 1)$–spanner has $\Omega(n^{1+1/k}/k)$ edges.

Distance Labelings and Metric Embeddings

The objective of distance labelings is to assign each node of a graph a label such that the distance (or an approximation thereof) between two nodes can be computed based on the corresponding labels only. Such labelings are used in the real world to a certain extent. For example, postal
addresses include countries, cities, and street names, using which we can get an estimate of how close two addresses are.

**Distance Labeling**

An \((\alpha, \beta)\)-approximate distance labeling scheme for a graph \(G = (V, E)\) is an assignment of labels to nodes \(L : V \rightarrow \{0, 1\}^*\) such that the estimated distance \(d(u, v)\) computed by the scheme from the labels \(L(u)\) and \(L(v)\) satisfies \(d_G(u, v) \leq \alpha \cdot d_G(u, v) + \beta.\)

Distance labeling schemes with short labels are derivable for highly regular graph classes, such as rings, meshes, and hypercubes. An interesting question is whether more general graph classes can also be labeled in this fashion. For general graphs, it is known that any distance labeling scheme must label some graphs with \(n\) vertices with labels of size \(\Omega(n)\). Even for planar graphs, some nodes must have labels of size \(\Omega(n^{1/3})\).

**Planer Graph Techniques**

A powerful approach when designing an algorithm is to use divide & conquer: we split the problem at hand into a bunch of smaller problems, solve these, and then combine their solutions. Ideally the sub-problems are completely independent such that the combination of their solutions is straightforward. In general, the sub-problems are not independent. It is then crucial to cut the problem into pieces with as few interdependencies as possible.

Road networks, although non-planar (the networks may have many bridges and tunnels), often also have structural properties that are similar to the ones for planar graphs. Most importantly, they appear to have small separators as well.

**Edge Orientability**

The edge set of a simple undirected planar graph \(G = (V, E)\) can be oriented (denoted by \(G = (V, \vec{E})\)) such that the out-degree of every vertex \(v\) satisfies \(\deg^+(v) = O(1)\). The upper bound \(G\) on the degree can be chosen to be \(O(1)\). Note that the nodes’ in-degrees \(\deg^-(v)\) remain \(G\) unbounded. Using this orientation, adjacency queries can be answered in constant time by inspecting both nodes. This technique can be seen as a labeling each node gets a label of size \(O(lg n)\) bits such that the adjacency of two nodes can be computed by looking at the two corresponding labels only. The result on edge orientability extends to minor-free graphs.

**Well-Separated Pairs**

For a graph \(G = (V, E)\) and a set \(A \subseteq V\), let \(G(A)\) denote the subgraph induced by \(A\). WSPD. For a graph \(G = (V, E)\), two sets of nodes \(A, B \subseteq V\) are separated if \(\max\{\text{diam}(G(A)), \text{diam}(G(B))\}\).
diam(G(B))) \cdot dG (A, B). For a parameter > 0, a well-separated pair decomposition of a graph G is a set of s pairs \( W = \{ \{ A_1 , B_1 \} , \ldots , \{ A_s , B_s \} \} \) such that for all pairs \( u, v \in V \)

\[
S (u, v) \in A_i \times B_i \cup B_i \times A_i ,
\]

\( i=1 \) and that for all \( i \in [s] \)

• \( A_i , B_i \subset V , \)

• \( A_i \cap B_i = \emptyset , \) and

• \( A_i \) and \( B_i \) are –separated.

**Shortest Paths**

Shortest paths should be of inherent interest to any lazy creature living on a sphere like the surface of planet Earth. Mikkel Thorup

Even in very primitive (even animal) societies, finding short paths and searching (for instance, for food) is essential. Alexander Schrijver. The distance between two nodes \( s, t \in V \) of a graph \( G = (V, E) \) is defined by the length of a shortest path. The objective is to find the shortest possible path that connects the source and the target.

**Single Source Shortest Path (SSSP) Algorithms**

The following outline is partially based on Ahuja, Magnanti, and Orlin’s book. Historical information is from. The Single Source Shortest Path (SSSP) problem asks for a shortest path from one node \( s \in V \) (called the source) to all other nodes in \( V \setminus \{s\} \). In particular, after running a single source shortest path algorithm, we know the distance from the source to all other nodes. After termination, each node \( u \in V \) is labeled with \( d(s, u) \). At the beginning \( d(s, s) = 0 \) and all other labels are set to \( \infty \).

SSSP algorithms iterate and assign tentative distance labels \( d(s, u) \) (upper bounds on the true distance \( d(s, u) \)) at each step. We distinguish label-setting, label-correcting, and other algorithms. In label-setting algorithms, each iteration “produces” one optimal label. This property does not hold for label-correcting algorithms, for which all labels are optimal after the final iteration only. Label-setting algorithms are restricted to non-negative lengths, but they often have a better worst-case time complexity than label-correcting algorithms. Also, if we are interested in one particular target node \( t \), it is possible to stop the algorithm as soon as the distance label for \( t \) has been produced.

**Label-Setting Algorithms**
The most famous label-setting algorithm is Dijkstra’s algorithm, see also or any algorithms textbook of your choice. The original implementation runs in time $O(n^2)$. According to an article on the history of combinatorial optimization, the same or a similar algorithm has been proposed independently by Leyzorek et al. [LGJ+ 57], Dantzig [Dan60], and Whiting and Hillier but it is known as Dijkstra’s algorithm. The algorithm starts a search at the source. At each step, among the nodes with tentative labels, the algorithm selects the node $u$ with the shortest tentative distance $d(s, u)$ and deems the label of $u$ as permanent. Then, the algorithm updates the labels of all the neighbors of $u$ if necessary. That is, if a neighbor $v \in \Gamma(u)$ has a tentative distance $d(s, v)$ larger than $d(s, u) + w(u, v)$, its label is decreased. The algorithm does not need to backtrack: once a label is finalized, it is correct and it can never decrease.

Another approach to efficient node selection is to use a priority queue, which is a data structure we use to manage the nodes. This data structure supports efficient operations to insert an element, to retrieve the minimum element, to delete the minimum element, and to decrease the value of an element in the queue. These are exactly the operations necessary for Dijkstra’s algorithm. Each update of a label involves a queue operation. If we restrict the range of the edge weight function with respect to integers (or floats), better bounds are possible by using special priority queues designed for the word RAM model.

**Component Hierarchy Algorithms**

There is an information-theoretic time lower bound of $\Omega(n \lg n)$. To circumvent this bound, sorting must be avoided. An SSSP algorithm is not actually required to sort the distances to all nodes. Indeed, the sorting bottleneck can be avoided. Thorup gives an $O(m)$ algorithm for undirected graphs with integer or floating point weights. He first constructs a component hierarchy (in linear time) and then revisits the nodes. Although the algorithm is theoretically best-possible, it appears to be hard to implement and not very efficient in practice. Hagerup generalizes the idea of the component hierarchy to directed graphs, for which his algorithm (using an analogue of minimum spanning trees for directed graphs) runs in time $O(m \lg \lg W)$. Actually, constructing the component hierarchy itself takes this time; an SSSP search using the hierarchy requires time $O(m + n \lg \lg n)$. Pettie and Ramachandran generalize the component hierarchy to real weights.

Due to the importance of the shortest path problem, researchers have been interested in parallel algorithms, I/O-efficient algorithms, algorithms for special graph classes such as planar graphs, sparse graphs, or Euclidean graphs, and in the expected performance (average-case analysis) of algorithms. Also, comparisons and experimental evaluation for practical purposes have been an important part of investigations. Point-to-point shortest path problems have been considered early on. In practice, point-to-point shortest path algorithms can be computed faster when bidirectional search is used. For road networks, if in addition to the graph the node coordinates
are known, A* heuristics based on geometry help to guide the search towards the target.

**All Pairs Shortest Path (APSP) Algorithms**

We consider the static version of the APSP problem, which is for example surveyed by Schrijver and Pettie. For the dynamic version of the problem, given a graph $G = (V, E)$, the All Pairs Shortest Paths (APSP) problem is to compute a shortest path, respectively the distance, between all pairs of nodes $s, t \in V$. Since there are $n^2 = \Theta(n^2)$ pairs of nodes for which the distance needs to be output, the time complexity $2$ must be at least $\Omega(n^2)$. Strong lower bounds are hard to prove. Nakamori give lower bounds for algorithms with restricted operations. Karger et al. shows that any algorithm based on path comparison must take time $\Omega(mn)$. For general algorithms, the gap between the lower and the upper bound is huge. Given an SSSP algorithm with running time $S(n, m, W)$, the straightforward approach to solve APSP is to run the SSSP algorithm for each node. This approach yields a running time of $O(nS(n, m, W))$. For non-negative weights, we may use Dijkstra’s algorithm, which yields time $O(mn + n^2 \log n)$. The algorithm of Johnson matches this bound also for negative weights (by using Dijkstra’s algorithm and the Bellman-Ford algorithm). For very sparse graphs with $m = O(n)$ edges, this is already best possible.

**Many Pairs Shortest Path (MPSP) Algorithms**

We may want to compute the shortest paths between pairs from a restricted subset of vertices $U \subseteq V$. For this, an APSP algorithm might be computing too much. Let $s := |U|$. For $s = \omega(1)$ pairs, the algorithm of Pettie and Ramachandran is currently the fastest method to compute exact shortest paths. Restricted graph classes. The algorithm by Cabello [Cab06] computes $s$-many distances in planar graphs in time $O(s^{2/3}n^{2/3} \log n + n^{4/3} \log^{1/3} n)$. The algorithm makes use of an efficient distance oracle for planar graphs.

**Many Pairs Approximate Shortest Path (MPASP) Algorithms**

Approximate distances between a subset of the nodes may be enough. For an overview of results, see. Similar to APASP, many algorithms use spanners to reduce the problem size. Elkin notes that the existence of an algorithm for constructing an $(\alpha, \beta)$-spanner with $O(n1+\zeta)$ edges ($\zeta > 0$) in time $O(T)$ implies the existence of an algorithm for the MPASP problem with $s$ sources with running time $O(T + s \cdot n1+\zeta)$.

**Shortest Path and Distance Queries**

Processing shortest path and distance queries in graphs can be seen as a generalization of the APSP problem and the MPSP problem, for which we do not know the pairs in beforehand.