Constant Time Queries for Energy Efficient Paths in Multi-Hop Wireless Networks

Stefan Funke Domagoj Matijevic Peter Sanders

Max-Planck-Institut f. Informatik

66123 Saarbrücken

Germany

***Abstract*— We investigate algorithms for computing energy efficient paths in ad-hoc radio networks. We demonstrate how advanced data structures from computational geometry can be employed to preprocess the position of radio stations in such a way that approximately energy optimal paths can be retrieved in constant time, i.e., independent of the network size. We put particular emphasis on actual implementations which demonstrate that large constant factors hidden in the theoretical analysis are not a big problem in practice.**

Keywords: Ad-Hoc and Sensor Networks, Routing, Power Control, Wireless LANs

I. Introduction

Ad hoc radio networks are an attractive way to quickly build a communication infrastructure without slow and expensive deployment of a cable backbone. Since many of the stations will be battery or solar powered, energy consumption becomes a major issue in such networks.

We use the following widespread model for energy consumption: The stations are defined by a set of n points in the plane. The energy consumption for communication between points p and q is assumed to be $\omega(p,q) = |pq|^\sigma$ for some constant $\sigma > 1$ where $|pq|$ denotes the Euclidean distance between p and q. In free space $\sigma = 2$ gives an exact physical model. Values $\sigma \in (2,4)$ can be used to approximate absorption effects [Rap96], [Pat00].

We are now looking for paths connecting arbitrary pairs of points that minimize energy consumption subject to the additional constraint that at most k hops are used. Limiting the number of hops accounts for distance independent energy consumption (e.g., for encoding and decoding signals) as well as for reliability and latency problems connected with paths that use an unbounded number of hops. For refinements of the model refer to Section IV. In our considerations we assume k to be a rather small constant.

This problem can be solved optimally in time $O(kn^2)$ using well known algorithms for computing shortest paths. However, this would be much too slow for all

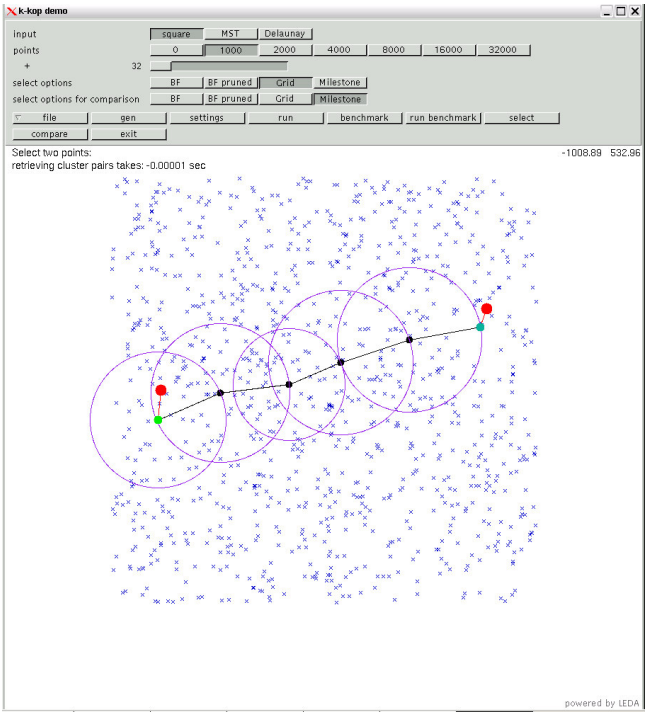

Fig. 1. Screenshot of our Simulation Program

but very small networks. In [FMS03] we have therefore developed an algorithm that produces paths that are within a factor $(1+\epsilon)$ from optimal in *constant time* independent of the size of the network. If k and ϵ are considered constants, the algorithm needs preprocessing time $O(n \log n)$ and space $O(n)$ for a lookup data structure. However, this theoretical algorithm has large hidden constant factors and it uses sophisticated data structures from computational geometry for which there is little experience with respect to their practicality.

The subject of the present paper is to help close this gap between theory and practice. We study a number of implementations of simple algorithms and new heuristics as well as a variant of the approximation scheme from [FMS03] but tuned for more practicability. Our presented solutions can be modified to provide for additional requirements like dynamic maintenance or fault-tolerance which both improve the quality of service.

Related Work

In 1998, Bambos [Bam98] reviewed developments in power control for wireless networks and emphasized the need for *minimum-power routing* protocols. Since then a vast amount of research has been conducted on the issue of energy-conservation in ad-hoc and sensor networks, see for example [JSAC02], [Pat00], [PRR01].

In the computational geometry community, Chan, Efrat, and Har-Peled [EH98], [CE01] have made several interesting observations for energy optimal paths with unbounded number of hops. They observe that it suffices to compute shortest paths in the Delaunay triangulation of the input points, i.e., optimal paths can be computed in time $O(n \log n)$. Note that this approach completely collapses for k hop paths because most Delaunay edges are very short. They also give a sophisticated $O(n^{4/3+\gamma})$ time algorithm for arbitrary monotone cost functions $\omega(p,q) = f(|pq|)$ where γ is any positive constant. For quadratic cost functions with offsets $\omega(p,q) = |pq|^2 + C$, Beier, Sanders, and Sivadasan reduce that to $O(n^{1+\gamma})$, to $O(kn \log n)$ for k-hop paths, and to $O(\log n)$ time queries for *two hop* paths using linear space and $O(n \log n)$ time preprocessing. The latter result is very simple, it uses Voronoi diagrams and an associated point location data structure.

II. Exact Algorithms for finding energy-minimizing k-hop Paths

Before we get to the actual algorithms let us give a more formal and abstract definition of our energy-minimizing k-hop path problem:

Given a set P of n points in $\mathbb{Z}^2$ and some constant k, report for a given query pair of points $s, t \in P$, a polygonal path $\pi = \pi(s,t) = v_0 v_1 v_2 \dots v_l$, with vertices $v_i \in P$ and $v_0 = s, v_l = t$ which consists of at most k segments, i.e. $l \leq k$, such that its weight $\omega(\pi) = \sum_{0 \leq i < l} \omega(v_i, v_i + 1)$ is minimized. By $\pi_{opt} = \pi_{opt}(s,t)$ we denote an optimal path from s to t under this criterion.

In the following we assume that the weight function ω is of the form $\omega(p,q) = |pq|^\sigma$ with $\sigma > 1$ (the case $\sigma \leq 1$ is trivial as we just need to connect s and t directly by one hop). For more general weight functions, in particular if we also have a constant, node-dependent offset like $\omega(p,q) = |pq|^\sigma + c_p$, we refer to Section IV for possible refinements of our presented algorithms.

A. The naive approach

The point set P together with the weight function ω induces the complete weighted graph $G(P, E, \omega)$ with vertex set P and edges $(v,w) \in E$ of weight $\omega(v,w)$, $\forall v \neq w \in P$. This graph has $n(n-1)/2$ edges and for a given query pair $s, t \in P$ we are looking for the shortest path $\pi_{opt} = \pi(s,t)_{opt}$ from s to t in G which uses no more than k edges.

This path π_{opt} can be easily computed by dynamic programming. Let $\pi(s,v)^{(i)}_{opt}$ denote the shortest path from the source node s to node v which uses no more than i edges. Clearly $\pi(s,v)^{(1)}_{opt} = sv$, $\forall v \in P - \{s\}$. $\pi(s,v)^{(i)}_{opt}$ is determined as $\pi(s,w)^{(i-1)}_{opt} v$ with w chosen such that $\omega(\pi(s,w)^{(i-1)}_{opt}) + \omega(w,v)$ is minimized.

The naive dynamic programming approach fills a table of dimension $n \times k$ using the above rules:

- $\forall v \in P$: $\pi(s,v)^{(1)}_{opt} \leftarrow sv$
- for $i = 2$ to k do
  - $\forall v \in P$:
    - compute $\pi(s,v)^{(i)}_{opt}$ by looking at all possible w, the concatenations $\pi(s,w)^{(i-1)}_{opt} v$ and their weights $\omega(\pi(s,w)^{(i-1)}_{opt}) + \omega(w,v)$

Clearly this algorithm has running time $O(k \cdot n^2)$ as we have to fill in a table of size $k \cdot n$ and determining the value of one cell costs $O(n)$ since we look at all possible $w \in P$. It is not hard to figure out that this approach only works for extremely small problem instances and even for those, it is rather slow as we get a quadratic behavior in n per query.

B. Neighborhood Pruning

One obvious improvement to the above algorithm is due to the observation that if we are interested in the energy-minimal k-hop path from s to t, points which are "far" away from the segment st cannot be of any use for the solution. So let D denote the distance between the query points, i.e. $D = |st|$. If we restrict our dynamic programming approach to all points $p \in P$ which have distance at most $\lambda \cdot D$ to the segment $|st|$ – we call this the *λ-neighborhood of st* –, what is the smallest value of λ such that we can still compute the optimal solution ? See Figure 2 for an example of λ-neighborhoods. It is not hard to see that if the optimal path π_{opt} leaves the region

which has distance at most $\lambda \cdot D$ to st, the sum of the Euclidean lengths of the segments of this path must be at least $2 \cdot D \cdot \sqrt{\lambda^2 + 1/4}$. And as the "optimal" strategy to chop a path of any given length into k pieces such that the overall energy is minimized, is to chop it into pieces of equal length, we get the following inequality

$$\frac{(2 \cdot D \cdot \sqrt{\lambda^2 + 1/4})^\sigma}{k^{\sigma-1}} \leq D^\sigma$$

which bounds λ in terms of the cost D^σ that are incurred when taking just one direct hop from s to t. So we get

$$\lambda_{\max} = \frac{\sqrt{k^{\frac{2\sigma-2}{\sigma}} - 1}}{2}$$

Therefore, if there are only few points in the neighborhood of the query points s and t (more precisely if there are only few points within distance $\lambda_{\max}|st|$), we first use a standard range query data structure from computational geometry to report all those points and run the naive approach only for those and can expect a reasonably fast query time, which is now only quadratic in the number of points in the neighborhood of s and t.

Cascaded Neighborhood Pruning: In the neighborhood pruning approach we have used the one-hop cost as an upper bound to limit the size of the neighborhood that still needs to be explored. Clearly, if we had a better upper bound (i.e. tentative solution) for the cost of getting from s to t within k hops, we could restrict the size of the neighborhood even further. How could such a better tentative solution be obtained? Well, we could start with a very small value for λ, even $\lambda = 0$ is viable, it just restricts the neighborhood to all points which lie on the segment st. We run our dynamic programming approach on that set of points and use the outcome to bound the maximal value of λ that we have to consider to guarantee the optimal solution is found. So this cascaded strategy could be implemented as follows:

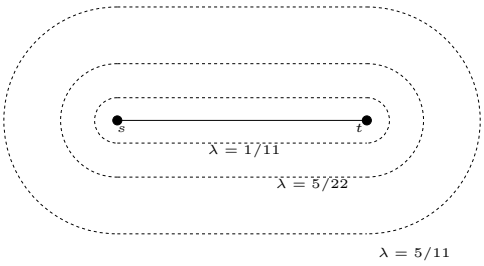

Fig. 2. λ-neighborhoods of a segment st

1) $\lambda \leftarrow 0.1$
2) upper $= |st|^\sigma$
3) while $\frac{(2 \cdot D \cdot \sqrt{\lambda^2+1/4})^\sigma}{k^{\sigma-1}} \leq D^\sigma$
   - compute using dynamic programming the optimal k-hop path w.r.t. the λ-neighborhood of st, update upper if necessary
   - $\lambda \leftarrow \lambda \cdot 2$

The procedure terminates as soon as it can prove that no larger neighborhood has to be inspected, which of course happens no later than after $O(\log \lambda_{\max})$ rounds. For dense point sets, this will turn out to be a lot more effective than the naive or simple neighborhood pruning strategy without cascading.

III. Approximate Algorithms for finding energy-minimizing k-hop Paths

The neighborhood pruning approach – though helpful for many problem instances – does not improve the worst-case running time of the dynamic programming approach as it might be the case that basically all the

points are in the neighborhood of the segment st and have to be inspected.

But if we relax the exactness requirement and only require approximate $(1+\epsilon)$ solutions, i.e. we are happy with paths $\pi(s,t)_{\text{app}}$ such that $\omega(\pi(s,t)_{\text{app}}) \leq (1+\epsilon)\cdot\omega(\pi(s,t)_{\text{opt}})$ for any $\epsilon > 0$ to be chosen from the user, we can do better. In fact, using *Grid Pruning* we can guarantee a logarithmic query time, when k, ϵ, σ are considered constants.

A. Grid Pruning

The idea of Grid Pruning is to place a grid over the neighborhood of the segment st and first report one *representative* in each of the grid cells (this can be done again using a standard geometric range query in time $O(\log n)$ per grid cell). The dynamic programming approach is then only performed on those representative points and the computed path is used as result of the computation. The smaller ϵ, the smaller the grid-cells will be, and hence the better approximation of the optimal path π_{opt} we get.

In fact, one can show (see [FMS03]) that putting a grid of cell-width $\alpha\cdot D/k$ with $\alpha = \frac{\ln 2}{2\sqrt{2}}\frac{\epsilon}{\sigma}$, the computed path $\pi_{\text{app}}(s,t)$ has cost at most $(1+\epsilon)\cdot\omega(\pi_{\text{opt}}(s,t))$.

The grid pruning algorithm looks as follows:

1) Put a grid of cell-width $\alpha\cdot|st|/k$ on the $\lambda_{\max}$-neighborhood of st with $\alpha = \frac{\ln 2}{2\sqrt{2}}\frac{\epsilon}{\sigma}$
2) For each grid cell C perform an orthogonal range query to either certify that the cell is empty or report one point inside which will serve as a representative for C.
3) Compute the minimum k-hop path $\pi(s,t)$ with respect to all representatives and $\{s,t\}$ using the dynamic programming approach.
4) Return $\pi(s,t)$

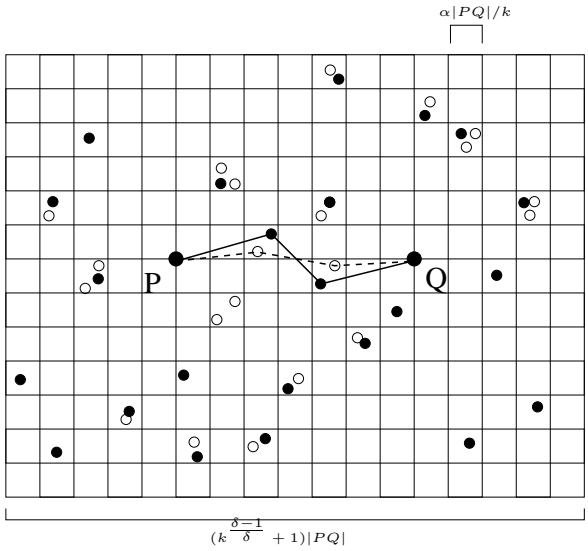

Fig. 3. 3-hop-query for P and Q: representatives for each cell are denoted as solid points, the optimal path is drawn dotted, the path computed by the algorithm solid

The gain compared to the previous methods is that we reduce the number of points to be considered to $O(\frac{\sigma^2\cdot k^{\frac{4\sigma-2}{\sigma}}}{\epsilon^2})$, irrespectively how many points there are in the neighborhood of st. So considering k, σ, ϵ constants, the query time becomes $O(\log n)$ due to the range queries, that have to be performed for each grid cell.

Please look at Figure 3 for a schematic drawing of how the algorithm computes the approximate k-hop path.

Cascaded GridPruning: Clearly, the same trick of looking at small neighborhoods of st first, which we have used to improve the neighborhood pruning approach, also works here. So first we only put the grid over a very small neighborhood and consider larger and larger neighborhoods until the required approximation

guarantee can be proven.

B. The Milestone Heuristic

For very dense point sets, there is another very simple heuristic, which uses the observation that in the "ideal" case, the segment st is divided into k subsegments of equal length. Clearly, if such a k-hop path can be obtained, it is the optimum path. So the *Milestone Heuristic* tries to approximate this "ideal" path by virtually placing the $k-1$ "ideal" radio stations $v_1, \ldots v_{k-1}$ on st. As these v_i are typically not in P, we perform for each of them a nearest neighbor query on the point set P and use the outcome as the replacement for v_i (each of these queries can be performed in $O(\log n)$ time). Nearest neighbor query data structures from computational geometry are by now standard in many software libraries and very space- and time-efficient implementations are available, e.g. in [LEDA]. So the algorithm looks as follows:

1) determine "ideal" hop positions $v_1, \ldots, v_{k-1}$
2) for each v_i perform a nearest neighbor query on P to obtain $v_i' \in P$
3) output $sv_1' \ldots v_{k-1}'t$ as the k-hop path

Unfortunately this approach can be fooled quite badly if the point set P is not equally distributed and there are large areas without any radio stations in the area between s and t.

C. Path Templates via Clustering

The best query scheme we have seen so far is able to answer a (s,t) query in O($\log n$) time (considering k, δ, ϵ as constants). Standard range query data structures were the only precomputed data structures used. Now we explain how additional precomputation can further reduce the query time. More precisely, we show how to precompute a *linear* number of k-hop paths, such that for every (s,t), a slight modification of one of these precomputed path templates is a $(1+\epsilon')$ approximate k-hop path and such a path can be accessed in constant time. Here $\epsilon' > 0$ is the error incurred by the use of these precomputed paths and can be chosen arbitrarily small.

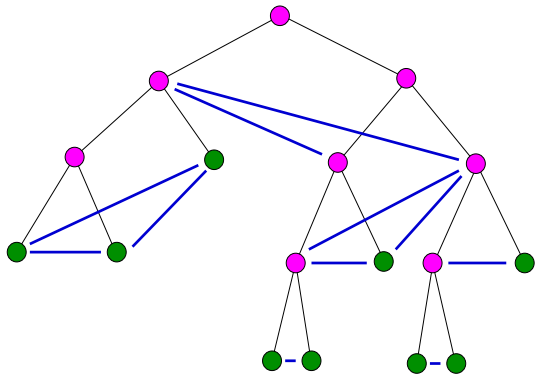

Fig. 4. Example of split tree with additional blue edges.

1) The Well-Separated Pair Decomposition: We will first briefly introduce the so-called *well-separated pair decomposition* due to Callahan and Kosaraju ([CK92]).

The *split-tree* of a set P of points in $\mathbb{R}^2$ is the tree constructed by the following recursive algorithm:

SplitTree(P)

1) if size(P)=1 then return leaf(P)
2) partition P into sets P_1 and P_2 by halving its minimum enclosing box $R(P)$ along its longest dimension
3) return a node with children (SplitTree(P_1), SplitTree(P_2))

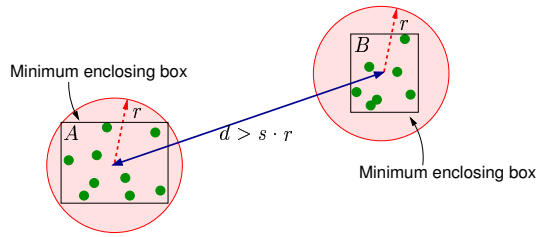

Fig. 5. Clusters A and B are 'well-separated' if $d > s \cdot r$.

Although such a tree might have linear depth and therefore a naive construction as above takes quadratic time, Callahan and Kosaraju in [CK92] have shown how to construct such a binary tree in $O(n \log n)$ time. With every node of that tree we can conceptually associate the set A of all points contained in its subtree as well as their minimum enclosing box $R(A)$. By $r(A)$ we denote the radius of the minimum enclosing disk of $R(A)$.

We will also use A to denote the node associated with the set A if we know that such a node exists.

For two sets A and B associated with two nodes of a split tree, $d(A, B)$ denotes the distance between the centers of $R(A)$ and $R(B)$ respectively. A and B are said to be *well-separated* if $d(A, B) > S \cdot r$, where r denotes the radius of the larger of the two minimum enclosing balls of $R(A)$ and $R(B)$ respectively. S is called the *separation constant*. Roughly, this means that the distance the centers of $R(A)$ and $R(B)$ is about the same as for any pair $a \in A$, $b \in B$.

In [CK92], Callahan and Kosaraju present an algorithm which, given a split tree of a point set P with $|P| = n$ and a separation constant S, computes in time $O(n(S^2 + \log n))$ a set of $O(n \cdot S^2)$ additional *blue* edges (A, B) for the split tree, such that

- the point sets associated with the endpoints of a blue edge are well-separated with separation constant S.
- for any pair of leaves (a, b), there exists exactly one blue edge (A, B) that connects two nodes on the paths from a and b to their lowest common ancestor $lca(a, b)$ in the split tree

The split tree together with its additional blue edges is called the *well-separated pair decomposition* $\mathcal{W}$ *(WSPD)*.

2) Application of the WSPD: Intuitively, the $\mathcal{W}$ encodes in linear space all $\Theta(n^2)$ distance relationships in the point set approximately. More precisely, for any query pair (s, t) there exists exactly one cluster pair $(A, B) \in \mathcal{W}$ with $s \in A, t \in B$ and $|\mathcal{W}| = O(n)$.

So we precompute for each of these $O(n)$ cluster pairs a good k-hop path between their respective centers (e.g. a $(1+\epsilon)$ path using the grid pruning strategy), such that at query time, for a given query pair (s, t), it only remains to find the unique cluster pair $(A, B) \in \mathcal{W}$ with $s \in A$, $t \in B$. We output the associated k-hop path replacing its first and last node by s and t respectively.

Since $s \in A$ and $t \in B$ and $d(A, B) > S \cdot r$, the precomputed path between the centers of c_A and c_B is 'almost' optimal for the query points s and t. In fact one can show formally that the returned path is a $(1 + \epsilon')$ approximation of the lightest k-hop path from s to t, where $\epsilon' > 0$ can be chosen arbitrarily by the user (this affects the required choice of the separation constant). See [FMS03] for the details.

How to retrieve the respective cluster pair (A, B) for

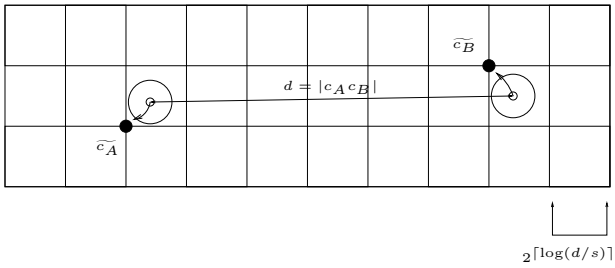

Fig. 6. Cluster centers c_A and c_B are snapped to closest grid points $\widetilde{c_A}$ and $\widetilde{c_B}$

a pair of query points (s,t)? The idea of our approach is to round the centers c_A, c_B of a cluster pair $(A,B) \in \mathcal{W}$ to canonical grid points $\widetilde{c_A}, \widetilde{c_B}$ and store the associated k-hop path in a hash table under the key $(\widetilde{c_A}, \widetilde{c_B})$, see Figure 6. As grid width we choose the next power of two of d_c/s, where $d_c = |c_A c_B|$. For a query pair (s,t) we have $d = |st| \approx d_c$ as $s \in A$, $t \in B$ and $(A,B) \in \mathcal{W}$. Hence, the same grid width as used for (c_A, c_B) can be determined from (s,t) (up to a factor of 2) and the path stored under the key $(\widetilde{c_A}, \widetilde{c_B})$ can be retrieved. See [FMS03] for the technical details on this procedure.

In fact in [FMS03] we have shown that for any (s,t) we can find exactly the respective cluster pair (A,B) in $O(1)$ time, but the constants hidden in the O-notation were quite huge (in the range of 10^6) mainly due to the fact that there might be many (even though $O(1)$) cluster pairs snapped to the same grid position. But for practical purposes, the k-hop path stored with any cluster pair (A',B') of those is good for us. Although it might not be true that $s \in A'$, $t \in B'$, we know that the cluster centers $c_{A'}, c_{B'}$ are close to s and t (otherwise they would not have been snapped to the same grid points) and therefore the respective k-hop path template is good for us.

IV. Refinements

In the following we will mention some refinements and extensions that are possible for the presented algorithms, some of which have already found their way into the current implementation.

A. Lazy Precomputation

In the path template approach as presented before, the idea was first to identify a collection of $O(n)$ source–target pairs (namely the centers of the clusters that are connected by a blue edge in the WSPD) and then precompute a good k-hop path for each of these pairs. In practice, it will turn out that identifying the blue edges can be done very quickly, and the really time-dominating step is the computation of the template paths (even when done using our $O(\log n)$ grid pruning approach).

But our data structure can easily be modified into a "lazy precomputation" scheme. So at precomputation time, only the blue edges are determind. At query time for a pair (s,t), we first identify the corresponding blue edge. If a template path has been stored for that edge already, we use it (and have spent $O(1)$ time only to answer the query). Only if no template path has been stored already, we compute one using the grid approach (making this query expensive, i.e. $O(\log n)$). Observe that if a similar query, i.e. a query (s',t') with s' near s and t' near t, arrives later, it will find the precomputed template path and can therefore be answered in $O(1)$.

B. Dynamization

All the data-structures that we have used are also – at least in theory – available in a dynamic version, where

updates can be performed in $O(\log n)$ time. Hence our whole construction could also be applied for moving and/or changing radio stations. Whether these dynamic versions of the algorithms are also of practical value, has to be shown or proven wrong by an experimental study. For more information on dynamic versions of the required data structures, we refer to [AM98], [AM00], [CK95].

C. Fault-tolerance

In many real-world applications reliability and quality of service (QoS) plays an important role. In particular, availability of the system has a very high priority. For our application this means that connections between two sites s and t should not be prohibited or become very expensive if some stations inbetween collapse. Therefore, it is very reasonable to provide for backup paths between the sites, i.e. if one or more stations of an energy efficient path between s and t become unavailable, there are other equally efficient paths already precomputed at hand. But this is easy to incorporate into our approach. For each blue edge of the WSPD we precompute instead of *one* template path *several* template paths which are all node-disjoint. These node-disjoint paths can be obtained as follows: First use the grid approach to compute the first energy-efficient k-hop path. Then remove all the used representative nodes from the grid cells that have been used in this path. If there are still other nodes left in the respective grid cells, use them to get another path, which looks very similar to the first one, but is node-disjoint from the latter. If some of the used grid-cells are empty, just run the brute-force algorithm on the remaining grid cell representatives. In this way, one can easily compute several path templates for each blue edge all of which are node-disjoint.

D. Startup-Costs

In our model as presented we restrict to a cost model where the required energy to transmit from p to q is $\omega(p,q) = |pq|^\sigma$. We can generalize this in the following manner: If the cost of transmitting from p to q is $|pq|^\sigma + C_p$ for a site dependent cost offset C_p, our result remains applicable under the assumption of some bound on the offset costs. In our grid pruning approach we would choose the cell representative as the node with minimum offset in the cell (this can be easily incorporated into the standard geometric range query data structures). The offset could model distance independent energy consumption like signal processing costs or it could be used to steer away traffic from devices with low battery power.

V. Implementation

All the algorithms mentioned in the previous sections were implemented using the LEDA library of data structures and algorithms ([LEDA]). We used the floating-point geometry kernel which represents points in the plane by two `double` coordinates. As range query structure we employed the LEDA datatype `point_dictionary` which allows range queries in time O($\log^2$) and nearest neighbor queries in O(n) worst-case time. But as these subroutines never dominated the running time in the respective algorithms where they were used, we did not put more effort into O($\log n$)

worst-case query time implementations.

A very critical issue was the use of an appropriate hashing datastructure for accessing the precomputed template paths. We are using the LEDA type `h_array` which hashes 32-bit integer values to some information domain. But our implementation requires to hash 4-tuples of 32-bit integer values. So we had to reduce the number of bits by a factor of 4. In our experiments the best choice for a hash function was to choose the 3rd to 10th least significant bits of each of these 4 integers and concatenate them to obtain the hash value for the 4-tuple. For other hash functions we tried, the number of collisions increased considerably and therefore accesses to the hashing table required going through a long list.

All algorithms were tested within an embedded simulation environment where data can be either read in or generated and then processed by our algorithms. Using a graphical user-interface, the different parameters and alternative algorithms can be selected and evaluated for running-time and quality of their produced solution. See Figure 1 for a screenshot of our simulation environment.

VI. Experiments

We conducted extensive experiments on different test data and using different parameters for our algorithms. All running times were measured on a low-end 700 MHz Pentium III with 256 MB of RAM. We used g++ 2.95.4 with the -O option under a Linux 2.4.19 system.

A. Benchmarks

Different test data sets were used to evaluate the quality of our algorithms. See Figure 7 for examples of the generated data.

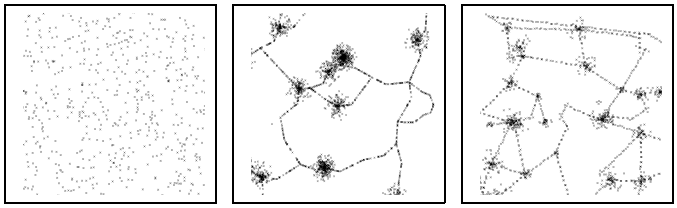

Fig. 7. Examples for test data: random (left), MST-based (middle) and Delaunay-based (right)

1) Random Data: Here we simply generated integer points uniformly at random in a square.

2) Simulated Real-World Data: As we had no real-world data available that could be put into a freely-available publication, we simulated the placement of radio stations along a road-network between cities. We had two simple algorithms to generate such data:

a) MST-based generation: We first generated a random set of points (the cities) and computed a Euclidean minimum spanning tree. For all the leaves of the tree inside the Convex Hull of the random set we added a new edge. Furthermore, we generated a cluster of points around every city and also put randomly some points along every edge (roads). At the end, we pruned sharp angles.

b) Delaunay-based generation: We first generated a random set of points (the cities) and computed the Delaunay triangulation. As we did not want to keep this ”triangular” road network, we removed some of the edges under the constraint that the remaining graph is still strongly connected. Then we assigned random weights to the cities and generated radio stations accordingly. Finally we generated some random stations along the remaining edges.

TABLE I

1000 POINTS RANDOMLY GENERATED; $k = 5$, $\sigma = 2$, $S = 5$, $\epsilon = 5$; QUERY TIME AND QUALITY

	WSPD	BF	BFp	Grid	Milestone
Av. Time	$8.0 \cdot 10^{-4}$	0.91	0.24	0.038	0.002
Max Time	$2.0 \cdot 10^{-3}$	1.45	1.24	0.080	0.01
Av. Rel. Err	15%	0	0	2.7 %	2.7 %
Max Rel. Err	49%	0	0	6.5 %	20 %
$\sigma_{\text{rel.err}}$	0.12	0	0	0.018	0.039

TABLE II

4000 POINTS RANDOMLY GENERATED; $k = 5$, $\sigma = 2$, $S = 5$, $\epsilon = 5$; QUERY TIME AND QUALITY

	WSPD	BF	BFp	Grid	Milest.
Av. Time	$5.66 \cdot 10^{-4}$	14.59	4.75	0.07	0.01
Max Time	0.003	24.63	14.46	0.099	0.01
Av. Rel. Err	16 %	0 %	0 %	2.6 %	0.5 %
Max Rel. Err	32.6 %	0 %	0 %	4.8 %	2.5 %
$\sigma_{\text{rel.err}}$	0.088	0	0	0.016	0.007

TABLE III

1000 POINTS FROM THE MST MODEL; $k = 5$, $\sigma = 2$, $S = 5$, $\epsilon = 5$; QUERY TIME AND QUALITY

	WSPD	BF	BFp	Grid	Milestone
Av. Time	$1 \cdot 10^{-4}$	1.193	0.937	0.009	0.0006
Max Time	$1 \cdot 10^{-3}$	1.63	4.03	0.01	0.01
Av. Rel. Err	14 %	0 %	0 %	3.6%	10.2 %
Max Rel. Err	38.7 %	0 %	0 %	14.4 %	35.9 %
$\sigma_{\text{rel.err}}$	0.123	0	0	0.047	0.114

TABLE IV

4000 POINTS FROM THE MST MODEL; $k = 5$, $\sigma = 2$, $S = 5$, $\epsilon = 5$; QUERY TIME AND QUALITY

	WSPD	BF	BFp	Grid	Milestone
Av. Time	$1 \cdot 10^{-4}$	18.6	10.1	0.024	0.011
Max Time	0.001	27.19	21.09	0.039	0.02
Av. Rel. Err	10.1 %	0 %	0 %	3.3 %	14.3 %
Max Rel. Err	20.5 %	0 %	0 %	8.1 %	33.7 %
$\sigma_{\text{rel.err}}$	0.048	0	0	0.026	0.109

B. Timings and Quality

In the following we are going to report timings and quality of the computed solutions for our different test data and varying problem sizes. For the precomputation of the template paths we chose a separation constant of $S = 5$ for the WSPD and $\epsilon = 5$ for the grid pruning subroutine. Even though in theory, this guarantees only a solution within a factor of 216 (!) of the optimal solution, in practice, the returned solutions were rather close to the optimum. For the used parameters of $k = 5$, $\sigma = 2$, $S = 5$, $\epsilon = 5$, in fact the returned solutions were not more than 20 % off the optimum on the average. See tables I, II, III, IV, V for the timing and quality results.

From the results you can see that the query time using the WSPD approach remains basically constant, independent of the problem size, which is not true for all other algorithms. In particular the brute-force variants suffer severely when increasing the problem size, but also the Milestone approach gets slower. The Grid approach also deteriorates a bit, but will saturate at some point (at least in theory). With regards to the quality, the brute force approaches are clearly the best since optimal, but also the Milestone Approach is not too bad. The results produced by the WSPD approach are mostly comparable to the Milestone and Grid approach but can be tuned by choosing different parameters as we will see later.

Of course, these very fast query times have their cost,

TABLE V

1000 POINTS FROM THE DELAUNAY MODEL; $k = 5$, $\sigma = 2$, $S = 5$, $\epsilon = 5$; QUERY TIME AND QUALITY

	WSPD	BF	BFp	Grid	Milestone
Av. Time	$4 \cdot 10^{-4}$	0.772	0.303	0.014	$5 \cdot 10^{-4}$
Max Time	0.002	1.13	1.28	0.03	0.01
Av. Rel. Err	17.2 %	0 %	0 %	5.7 %	10.7 %
Max Rel. Err	35 %	0 %	0 %	56 %	57 %
$\sigma_{\text{rel.err}}$	0.101	0	0	0.12	0.134

TABLE VI

TIME/SPACE FOR PREPROCESSING ON 1000 RANDOM POINTS, $k = 5$, $\sigma = 2$ AND VARYING ϵ AND S

	# templ.	time (s)	avg.err	max err
$\epsilon = 10, S = 4$	7813	57.7	23%	58 %
$\epsilon = 10, S = 5$	12042	98.0	22%	47 %
$\epsilon = 10, S = 7$	21200	187.3	14%	36 %
$\epsilon = 10, S = 11$	46148	458.8	12%	29 %
$\epsilon = 5, S = 4$	8287	145.7	17%	37 %
$\epsilon = 5, S = 5$	12004	230.6	15%	27 %
$\epsilon = 5, S = 7$	21924	559.8	12%	34 %
$\epsilon = 5, S = 11$	46236	1446.5	6 %	13 %
$\epsilon = 2, S = 4$	7925	433.91	22 %	47 %
$\epsilon = 2, S = 5$	11724	712.31	9 %	28 %
$\epsilon = 2, S = 7$	22126	1606	9 %	31 %
$\epsilon = 2, S = 11$	43347	3875	5 %	24 %

both in terms of time for the precomputation as well as in terms of the space required to store the template paths. For this purpose we look again at the example of 1000 random points but now vary both ϵ (the parameter used for the grid approach when computing the template paths) as well as S (the separation constant for the WSPD). See table VI for the results. Apart from the size of the precomputed structure and the preprocessing time we show the average and maximal relative error that was incurred by the precomputed paths for 30 random k-hop queries.

Clearly, the more time and space one is willing to invest into computing good path templates, the better results one gets for the queries. We remind that all the precomputation can be done in a lazy fashion as explained in Section IV, so the precomputation time would only consist of the time required to construct the WSPD, which is neglectable. If a query is "new" in a sense that no similar query has been performed before, the respective path template will be computed, so the set of path templates is built up one by one during the queries. Once all path templates have been constructed, the data structure behaves exactly as its counterpart where all precomputation has taken place before the queries.

TABLE VII

TIME FOR PREPROCESSING ON 1000 RANDOM POINTS, $\sigma = 2$, $S = 5$,$\epsilon = 5$, VARYING k

	WSPD		
	pre. time(s)	avg. err	max err
$k = 2$	14.6	6.1 %	13.8 %
$k = 4$	74.8	15.9 %	41.6 %
$k = 8$	530	25 %	41.1 %
$k = 16$	3471	29.2 %	55.8 %

The choice of k – the number of allowed hops – also affects the running time of the grid pruning approach and therefore of the preprocessing step. See table VII.

As larger values for k require a finer grid, the running time of the precomputation grows rapidly. To keep the

TABLE VIII

TIME FOR PREPROCESSING ON 1000 RANDOM POINTS, $k = 5$, $S = 5$,$\epsilon = 5$, VARYING σ

	pre. time (s)	avg. err	max err
$\sigma = 2$	232	14 %	30 %
$\sigma = 3$	524	30 %	60 %
$\sigma = 4$	817	41 %	75 %

quality of the solution, we would have had to increase the value for S as well to accommodate for the finer granularity of the solution.

As mentioned in the introduction, even though setting $\sigma = 2$ models the exact, free-space energy consumption, in practice people use larger values $\sigma \in [2, 4]$ to account for absorption effects etc. As σ also affects the running time of the grid pruning approach, we give experimental data for varying σ in Table VIII.

It turns out that higher values for σ induce a considerably higher precomputation since the grid size chosen by the grid pruning algorithm is smaller. But still, the quality deteriorates, as with the larger exponent in the cost function, even small perturbations might increase the cost considerably. So to keep the same error bounds, a smaller value for ϵ and/or a larger value for S would have to be used.

VII. CONCLUSIONS

We have demonstrated that near energy optimal paths can be queried very efficiently even in large radio networks. If the network is not too large, even slowly changing networks can be accommodated. Nevertheless, many questions remain how such a technique could be used in real networks.

As long as the network is static, rather large networks could be handled. For small networks the precomputed tables could even be replicated on all nodes. For large networks the hash table can be distributed over the network. If paths are used for a long time (seconds) compared to the time needed for querying a path (milliseconds) even a centralized server for connection queries would be feasible. In that case even occasional updates for inserting, deleting, or moving stations would be feasible.

Finite maximum ranges can be accommodated easily by ignoring all connections that exceed this range in the path computations.

Distributed implementations that can accommodate large and dynamic networks are a challenge beyond the scope of this paper however. Still, at least simple grid-based data structures may be helpful here and can use similar arguments as in our centralized algorithm.

Contention of several routes that use the same frequency bands at the same time are an issue not directly accessed by our shortest path model. However, for $\sigma = 2$, our cost model minimizes the sum of the areas covered by the transmitters used in a path. This can have an indirect positive effect on contention.

Minimal total energy consumption does not guarantee fairness, i.e., it might happen that one station is used so often that its batteries are quickly drained. This effect can be mitigated in several ways. For example, rather than storing fixed routes, we can simple store areas (e.g. squares) where relais stations should be located. Any

combination of points in these relay areas will yield an energy efficient path. At least in densely populated areas one can then balance energy consumption by picking random stations in each relay area. One can even explicitly take energy reserves or other priorizations into account.

REFERENCES

[AM00] S. Arya and D. M. Mount:*Approximate range searching*, Computational Geometry: Theory and Applications, (17), 135-152, 2000

[AM98] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, A. Wu: *An optimal algorithm for approximate nearest neighbor searching*, Journal of the ACM, 45(6):891-923, 1998

[Bam98] N. Bambos: *Toward Power-Sensitive Network Architectures in Wireless Communications: Concepts, Issues, and Design Aspects*, IEEE Personal Comm., Vol 5, June 1998

[BSS02] R. Beier, P. Sanders, N. Sivadasan: *Energy Optimal Routing in Radio Networks Using Geometric Data Structures* Proc. of the 29th Int. Coll. on Automata, Languages, and Programming, 2002.

[BKOS] M. de Berg, M. van Krefeld, M. Overmars, O. Schwarzkopf: *Computational Geometry: Algorithms and Applications*, Springer, 1997

[CK92] P.B. Callahan, S.R. Kosaraju: *A decomposition of multi-dimensional point-sets with applications to k-nearest-neighbors and n-body potential fields*, Proc. 24th Ann. ACM Symp. on the Theory of Computation, 1992

[CK95] P.B. Callahan, S.R. Kosaraju: *Algorithms for Dynamic Closest Pair and n-Body Potential Fields*, Proc. 6th Ann. ACM-SIAM Symp. on Discrete Algorithm, 1995

[CW79] J.L. Carter and M.N. Wegman. Universal Classes of Hash Functions. *Journal of Computer and System Sciences, 18(2):143–154, 1979*

[CE01] T. Chan and A. Efrat. Fly cheaply: On the minimum fuel consumption problem. *Journal of Algorithms*, 41(2):330–337, November 2001.

[EH98] A. Efrat, S. Har-Peled: *Fly Cheaply: On the Minimum Fuel-Consumption Problem*, Proc. 14th ACM Symp. on Computational Geometry 1998.

[FMS03] S. Funke, D. Matijevic, P. Sanders: *Approximating Energy Efficient Paths in Wireless Multi-Hop Networks*, Proc. of 11th European Symposium on Algorithms 2003 (ESA), number 2832 in LNCS, pages 230-241. Springer.

[GW02] A. Goldsmith, S.B. Wicker (eds.): *Special Issue: Energy-Aware Ad Hoc Wireless Networks*, IEEE Wireless Comm., Vol.9, Aug. 2002

[JSAC02] C.E. Jones, K.M. Sivalingam, P. Agrawal, J.C. Chen: *A Survey of Energy-Efficient Network Protocols for Wireless Networks*, Wireless Networks, Vol. 7, July 2001

[LEDA] K.Mehlhorn, S.Näher: *LEDA: A platform for combinatorial and geometric computing*, Cambridge University Press, 1999

[MN90] K. Mehlhorn, S. Näher: *Dynamic Fractional Cascading*, Algorithmica (5), 1990, 215–241

[Pat00] D. Patel. Energy in ad-hoc networking for the picoradio. Master's thesis, UC Berkeley, 2000.

[PRR01] C. Petrioli, R.R. Rao, J. Redi (eds.): *Special Issue: Energy-Preserving Protocols*, Mobile Networks and Applications, Vol.6, June 2001

[Rap96] T. S. Rappaport. *Wireless Communication*. Prentice Hall, 1996.

[ThoZwi01] M.Thorup and U.Zwick. Approximate Distance Oracles *Proc. of 33rd Symposium on the Theory of Computation 2001.*