Practical compact routing on random unit disk graphs

Craig Gotsman · Kai Hormann

Abstract

We describe a simple and practical algorithm for compact routing on connected random unit disk graphs. Using a recursive nested dissection of an n-vertex graph based on compact and balanced vertex separators, we construct routing tables with an average of $O(\log^2 n)$ entries per vertex in a preprocessing step. The routing tables then support handshaking-based routing, where the handshaking can be implemented similarly to a DNS lookup. Our routing algorithm is *guaranteed* to deliver on the graph, while incurring moderate stretch. We describe a basic version of the algorithm that requires modifiable headers and a more advanced version which eliminates this need and incurs less stretch.

Citation Info

Journal
ACM Transactions on Sensor Networks

Volume
22(1), January 2026

Article
1, 23 pages

DOI
10.1145/3778347

1 Introduction

Compact routing arises in communication networks where packets are to be sent between two arbitrary nodes of the network along a short path or *route*. Call these two nodes s and t, the source and target nodes. To facilitate this, each node v, or *router*, of the network contains a *routing table* that indicates, when a data packet destined for t arrives at v, which of the routers connected to v will be the best one to forward the packet to. Obviously, these routing tables should be designed such that they are *correct*, so always *deliver*, that is, a packet is always guaranteed to reach its destination t and never get "stuck" at any intermediate router, or, more generally, end up routed in an infinite loop. Additionally, the routing tables should be *compact*, requiring only $o(n)$ storage per router, where n is the number of nodes, and generate routes with low *stretch*, where the stretch measures the ratio between the lengths of generated route and the shortest possible route between s and t in the network. There is typically a tradeoff between the routing table storage required by any such scheme (sometimes called the *state*) and the resulting stretch of the routes. Other relevant, but less important, parameters are the computation time required to build the routing tables (which is typically done in a preprocessing step), the computation time required to determine the next router from the routing table, and the size of the header that each packet must contain with information about the destination node t. Historically, compact routing algorithms preferred the header not to be modified during the routing, but more recent algorithms using so-called *modifiable* or *rewritable* headers permit this in order to improve the performance of the routing algorithm. Moreover, some schemes achieve smaller routing tables or lower stretch by permitting a *handshaking* mechanism at the beginning of each routing procedure.

While it is important to design compact routing schemes for general networks, which are modeled as graphs, there are important special cases where the task is easier, and the storage-stretch tradeoff is more manageable. This typically is possible in *sparse* graphs, those where the number of edges is far less than the maximal $O(n^2)$ possible. In this paper we treat graphs which arise in sensor networks, Internet-of-Things and radio and mobile communications scenarios, where nodes are scattered randomly in the plane (or a higher-dimensional Euclidean space) and can communicate with only a subset of the nodes geographically close to them. The underlying mathematical model is the *unit disk graph* (UDG), where a node is connected to all others within a fixed radius. Among others, this models well wifi and bluetooth communication networks.

UDGs have an important property that we will take advantage of: the existence of compact balanced separators [8, 12]. For plane UDGs, this means that given such a graph G with vertex set V of size n and edge set E of size m, there always exists $S \subseteq V$, a compact subset of the vertices, such that removal of the vertices of S and the edges incident on them *separates* the graph into two disjoint components with vertex sets U_0 and U_1, both of size at most $2n/3$. According to Fox and Pach [12], the size of S is $O(\sqrt{m})$ and according to Carmi et al. [8], the size is $O(\sqrt{(m+n)\log n})$. Despite the latter being a little worse than the first, Carmi et al. [8] show that S may always be taken as the endpoints of a set of edges intersected by a straight line through the "midpoint" of the graph and may be found in $O(n)$ time using a simple "line-sweep" algorithm. They furthermore show that if only axis-parallel lines are considered, components of size at most $4n/5$ may

still be obtained. In contrast, the separators of Fox and Pach [12] are more elaborate and require a more complicated algorithm. As we will see later, we operate in a setting where $m = O(n \log n)$, which guarantees favourable connectivity properties. Thus, the size of these separators is $O(\sqrt{n} \log n)$.

The existence of such separators in UDGs facilitates an efficient "divide-and-conquer" strategy, called *nested dissection*, which is run on the graph, recursively partitioning it into smaller and smaller pieces, represented as a *separator tree*. The smallest pieces of the graph, at which the recursion terminates, are called *clusters*. These are the leaves of the separator tree. Our key observation is that it is possible to use the separator tree to build compact routing tables which are guaranteed to deliver, generating routes with moderate stretch. In a nutshell, the routing tables guide the routing process such that the packet advances towards a given target vertex t by repeatedly approaching deeper and deeper separators between it and t. Once a cluster containing t is reached, the routing concludes using a conventional routing method within the cluster.

1.1 The random UDG model

As mentioned, UDGs are used to model the physical aspects of many communication scenarios. Of special interest are UDGs which arise from a simple probabilistic model, in which no specific structure of the graph is assumed. In this model, the graph vertices are assumed to be an instance of a spatial random process, in which the n vertices (nodes) are located in a square of size $L \times L$, scattered with a uniform distribution at a given density $\rho = n/L^2$, specifying the average number of vertices per unit area. Thus, for large L, the number of vertices per unit area follows the Poisson distribution with parameter ρ. As usual, two vertices are connected by an edge of the UDG if the distance between them is less than unit. This model of a random UDG with vertex density ρ has been studied extensively in the literature (e.g., [14, 27]), both analytically and experimentally. It is well known that as $L \to \infty$, there is a critical density ρ_P, called the *percolation threshold*, such that the graph contains an infinite connected component (the so-called "giant component") with probability close to 1 ("almost certainly"), if and only if $\rho \geq \rho_P$. There is also a critical density ρ_C, called the *connectivity threshold*, such that the graph is connected with probability close to 1, if and only if $\rho \geq \rho_C$. It has been shown empirically that $\rho_P \approx 2.87$ [14] and $\rho_C = c \log n$ for some constant c. Xue and Kumar [31] found empirically that $c \leq 1.64$, but it is commonly believed to be much less, probably $c \approx 0.32$ [17]. In the sequel, we will use the equivalent quantity *edge density* $\alpha = m/n$, which is half the average number of neighbours per vertex. Thus $\alpha_P = \frac{1}{2}\pi\rho_P \approx 4.51$ and $\alpha_C = \frac{1}{2}\pi\rho_C \approx 0.5 \log n$. As we mention later, we will assume a random model where $\alpha = O(\log n)$ with $\alpha \geq \alpha_C$, so that connectivity is essentially guaranteed.

1.2 Related work

Compact routing has been investigated since the 1980's, but the first landmark paper, providing a universal scheme for compact routing on general graphs, was by Thorup and Zwick [30] in 2001. They introduce a stretch-3 method with optimal $O(\sqrt{n})$ storage (up to logarithmic factors), establish lower bounds on the storage-stretch tradeoff, and further propose handshake-based variants of their method with presumably optimal tradeoff. These methods remained somewhat theoretical until 2010, when the practical S4 implementation of Mao et al. [24] (based on [30]) appeared. A survey of the progress until 2007 was made by Dom [10].

Better results can be achieved for graphs that admit compact and balanced separators, e.g., for planar graphs [1, 13, 23, 29]. The existence of such separators for UDGs was first discovered by Alber and Fiala [3]. Yan et al. [32] proved another compact separator theorem for UDGs, which enables an $O(\log n)$ spanning tree cover with stretch 3 and can be used to generalize the routing method of Gupta et al. [16] to UDGs. Improvements on separator theorems for UDGs have been obtained over the years, culminating in the results of Fox and Pach [12] and Carmi et al. [8], who also prove a bound on the size of the separator. Kaplan et al. [18] describe a routing method for UDGs based on the existence of a *well-separated pair decomposition* of the graph and modifiable headers with a polylogarithmic number of bits. This provides an explicit tradeoff between state and stretch, as follows: for any $\epsilon > 0$, there is a routing scheme with stretch $1+\epsilon$ using routing tables of size $O(\epsilon^{-5} \log^2 D \log^2 n)$ bits, where D is the Euclidean diameter of the graph. Most recently, Mulzer and Willert [25] construct, for any $\epsilon > 0$, a routing scheme that achieves stretch $1+\epsilon$ and storage of $O(\log D \log^3 n / \log\log n)$, where the constant in the O-notation depends on ϵ.

Related to compact routing is a body of work on *geographic routing*, which uses only *local* information retrieved from the immediate geometric neighbours to make the routing decision (i.e., to which neighbour

of the vertex to proceed). The simplest *greedy* routing method is to proceed to the neighbouring vertex that is the closest in Euclidean distance to the target. An advantage of these greedy methods is that they are nearly stateless and thus highly scalable and suited for large wireless sensor networks. However, as with compact routing, the main challenge of these local methods is to guarantee *delivery*, namely that they not get stuck in a local minimum, caused by network "holes", for example. A classic result of Bose and Morin [5] states that Euclidean coordinates can support greedy routing on the Delaunay triangulation of any set of points in the plane.

Although geographic routing was originally based on the information derived from the actual coordinates of the vertices in space (typically the two-dimensional plane), more recent methods generate their own "virtual" coordinates on which the routing is performed. A series of results show that *any* 3-connected planar graph admits a *greedy embedding*, namely, may be embedded in the plane in a way that Euclidean distance-based greedy routing delivers [4, 9, 21, 26].

For UDGs or general graphs, it is necessary to embed in a higher-dimensional space. Indeed, a greedy embedding of a UDG on n vertices using $O(1)$ coordinates can incur stretch of $O(n)$ [20]. Using more coordinates would reduce the stretch, and if these new coordinates are succinct, their storage could be considered equivalent to a compact routing table. The method of Flury et al. [11] shows how to embed a UDG on n vertices into a space of dimension $O(\log^2 n)$, where each coordinate consumes $O(\log n)$ bits, thus equivalent to a routing table of size $O(\log^3 n)$ bits per vertices. The resulting greedy routing (using a *min-max* distance function) guarantees delivery and constant stretch. An extension of the method to general graphs guarantees $O(\log n)$ stretch. Caragiannis and Kalaitzis [7] show that this is almost the best that can be hoped for, as any greedy embedding of a general n-vertex graph in polylog(n) dimensions necessarily incurs a stretch of $\Omega(\log n/\log\log n)$. They also show that for constant stretch $\Omega(n)$ dimensions are required.

1.3 Our contribution

While there has been a lot of theoretical work on the topic of compact routing on general graphs, and also the special case of UDGs, none of the published papers, except Mao et al. [24] (for the general case), report implementation details, and we are not aware of any independent implementations of these ideas. This could be because they rely on quite sophisticated data structures and algorithms, which are difficult to implement in practice, or because the asymptotic complexity bounds contain large constant factors, which limit the effectiveness of the methods on moderately-sized graphs.

Thus, our main contribution is the description of a relatively simple, yet effective, method for compact routing on connected random UDGs, which is proven to *always deliver* (see Section 3.4). The only complicated element of the algorithm is the computation of vertex separators, but this is also relatively easy in practice, since efficient implementations can be found in many publicly available software packages, and the method of Carmi et al. [8] is very easy to implement. Since we assume a random UDG model with edge density $\alpha = O(\log n)$ above the connectivity threshold α_C, we can show (see Section 3.2) that the average number of entries in a vertex routing table is $O(\log^2 n)$. Note that each entry consists of $O(\log n)$ bits.

We also report on the experimental results (see Section 4) of our implementation, which can be made available for public use. Our experiments confirm that the size of the routing tables is $O(\log^2 n)$ in practice and that our method generates routes with an average stretch of less than 1.2 and a maximal stretch of at most 4. Alas, we were not able to prove these bounds on the stretch.

2 The basic idea

In the following subsections, we outline the basic version of our routing method, involving a nested dissection of the graph. Its implementation as a set of compact routing tables requires the use of an initial handshaking step to create a header with information about the target t that is attached to the packet. Moreover, it requires the header to be modifiable, in order to "remember" some information during the routing process. We later show how to implement a more sophisticated version of the algorithm, which gets by with a non-modifiable header and even improves the stretch relative to the basic method.

2.1 Nested dissection

Given a connected graph $G(V,E)$ with vertex set V of size n and edge set $E \subset V \times V$ of size m, we construct a nested (binary) dissection [15] of G, represented by a binary tree T (see Figure 1). Denote by $G(U)$ the

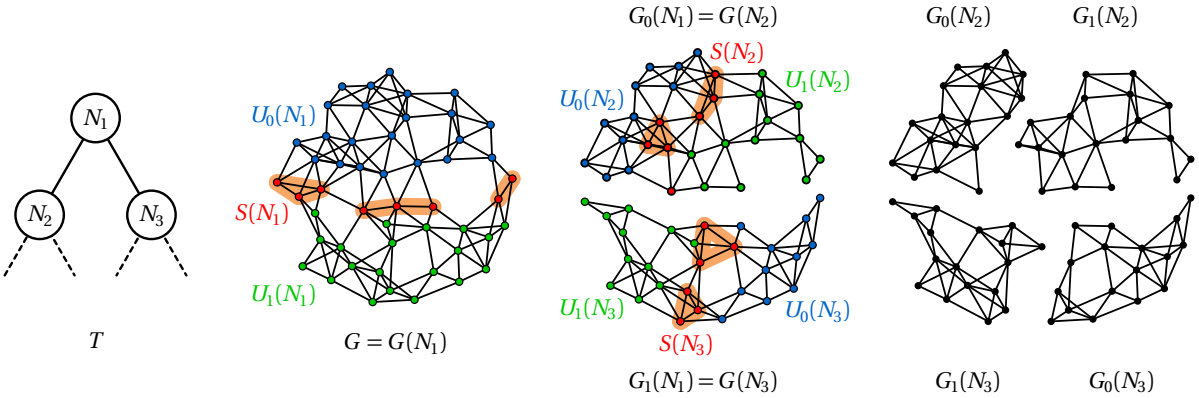

Figure 1: The root node N_1 of the separator tree T represents the given graph $G = G(N_1)$ and a separator $S(N_1)$ (red). The separator is chosen such that G, after removing the vertices of $S(N_1)$ and all edges adjacent to them, is split into two connected components with vertices $U_0(N_1)$ (blue) and $U_1(N_1)$ (green) and may itself be partitioned into several disjoint components (orange). The two children N_2 and N_3 of N_1 represent the subgraphs $G_0(N_1)$ and $G_1(N_1)$ of G with vertices $U_0(N_1) \cup S(N_1)$ and $U_1(N_1) \cup S(N_1)$, respectively. The dissection process keeps splitting these subgraphs in the same way, until they become small enough.

subgraph of G defined by the subset of vertices $U \subset V$, inheriting the edges between vertices of U. The root of T represents G and contains a vertex separator S that splits G into two subgraphs, and this is recursively true for all nodes of T. More precisely, a node N of T represents a connected subgraph $G(N) = G(V(N))$ of G with vertices $V(N) \subset V$ and a separator $S(N) \subset V(N)$, such that $V(N) \setminus S(N)$ consists of exactly two connected components, with vertex sets $U_0(N)$ and $U_1(N)$, respectively.

Let $V_0(N) = U_0(N) \cup S(N)$ and $V_1(N) = U_1(N) \cup S(N)$, and denote by $G_0(N)$ and $G_1(N)$ the subgraphs of G with vertex sets $V_0(N)$ and $V_1(N)$, respectively. To facilitate routing, in this node N we store the list of separator vertices $S(N)$ and a set of graph distances (i.e., hop counts) between $S(N)$ and the vertices $V_0(N)$ *within* $G_0(N)$ and between $S(N)$ and the vertices $V_1(N)$ *within* $G_1(N)$, respectively. In some cases, it is beneficial to further partition $S(N)$ into a number of disjoint *components* $C \subseteq S(N)$ with at most γ vertices each, for some global constant γ. To be precise, we make sure that a separator with τ vertices is split into *at most* $\lceil \tau/\gamma \rceil$ components, each with *at most* γ vertices. As we will see below, it is best to use $\gamma = \Omega(\sqrt{n})$, so that a balanced separator with $O(\sqrt{n} \log n)$ vertices is split into no more than $O(\log n)$ components. Instead of one distance, we then consider the *vector* of distances between the vertices and these components. The two children of N then represent $G_0(N)$ and $G_1(N)$ respectively. We say that $s, t \in V(N)$ are *separated* by $S(N)$, if $s \in V_0(N)$ and $t \in V_1(N)$, or vice versa. We further say that $S(N)$ is a *strong separator* of s and t, if $s \in U_0(N)$ and $t \in U_1(N)$, or vice versa (namely, both s and t are not in $S(N)$).

Ultimately, the recursion terminates when the subgraph of G represented by a node of T is small enough in some meaningful sense. The simplest way to do this is to terminate once the level of the recursion exceeds some specified depth, as the size (i.e., number of vertices) of a subgraph at depth k will typically scale like $2^{-k}n$. More sophisticated termination criteria could take into account the actual size of the subgraph, or its graph-theoretic diameter. We chose to use the diameter criterion, so the leaves of T represent small subgraphs G' of G with bounded diameter $\mathrm{diam}(G') \leq \delta$ that we call *clusters*. The global constant δ does not depend on n and is typically quite small (e.g., $\delta \leq 5$).

Note that as the dissection generates subgraphs having the vertices of the separator in common, each vertex v of G may belong to more than one cluster. We informally call the separators at nodes close to the root of the tree T *high-level* separators and the ones close to the leaves *low-level* or *deep* separators. We say that a node N of T and its separator $S(N)$ are *associated* with all vertices $v \in V(N)$. Vice versa, each vertex $v \in V$ is contained in a subset of the leaves of T, so we may associate with v the *separator subtree* T_v rooted at the root of T whose leaves contain v, but *without* the leaves themselves, as well as all the nodes of T_v and the separators represented by these nodes.

2.2 Vertex separators

Nested dissection depends on the existence of compact balanced separators and an efficient algorithm to compute such separators. The celebrated theorem of Lipton and Tarjan [22] guarantees that every n-vertex

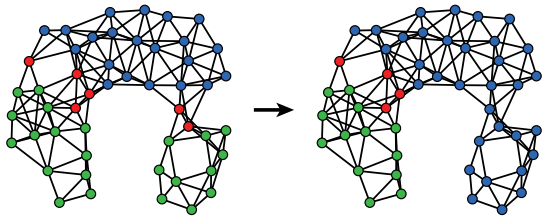

Figure 2: If the separator splits the graph into more than two connected components (left), then we combine components until only two are left (right).

planar graph contains a separator of size $O(\sqrt{n})$, such that the size of each separated component is between $n/3$ and $2n/3$. Similar theorems exist also for UDGs [8]. A simple and common way that we use to generate good separators, which usually gives components with this property, is the spectral method [28], where the so-called *Fiedler eigenvector* of the graph Laplacian matrix is used. This vector is negative on approximately half the vertices and the edges connecting the negative vertices with positive ones are a compact *edge cut* of G. This implies that the positive (or negative) endpoints of these edges are a compact vertex separator of the graph. More elaborate separation methods exist (see survey by Buluç et al. [6]), some implemented quite efficiently in the METIS software package [19]. Alternatively, it is possible to use the simple $O(n)$ geometric methods of Carmi et al. [8] to generate compact and balanced separators for UDGs.

Of additional interest are the connectivity properties of the two components. As we will see later, for our compact routing algorithm, it is essential that the two components produced during the separation of a connected graph are themselves connected. While no algorithm actually guarantees that, we argue that in the random UDG model where the edge density is above the connectivity threshold ($\alpha \geq \alpha_C$), since the separator has a simple geometric shape, the edges removed by the separation will not be damaging and each component will remain above the threshold, thus preserving the connectivity property on both sides of the separator. If this does not happen, connected components can be merged together to form just two components (see Figure 2). Should this damage the balance too much, the nested dissection may be terminated at that point (and the unseparated component becomes a leaf cluster).

2.3 The separator tree

Algorithm 1 describes the recursive function NESTEDDISSECTION that constructs the separator tree T for a graph $G(V,E)$. Each interior node N of T contains the components of a separator, the distances from the subgraph vertices to each separator component, and pointers to the nodes representing the two subgraphs

Algorithm 1 Recursively dissect the graph $G = G(V,E)$ and build T

```
1:  function NestedDissection(G, γ, δ)
2:      create node N                                   ▷ node N of T that represents G
3:      N.vertices := V(G)                              ▷ store the vertices of G
4:      if diam(G) > δ then                             ▷ dissect G only if its diameter is greater than δ
5:          (U_0, U_1, S) := Separate(G)                ▷ split the vertices of G into disjoint sets U_0, U_1, and S,
6:          N.component := Partition(S, γ)              ▷ partition S into components, each with at most γ vertices
7:          G_0 := G(U_0 ∪ S)                           ▷ "left" subgraph of G with vertices V_0 = U_0 ∪ S
8:          G_1 := G(U_1 ∪ S)                           ▷ "right" subgraph of G with vertices V_1 = U_1 ∪ S
9:          for C ∈ N.component do                      ▷ compute shortest path distances in both subgraphs
10:             for u ∈ U_0 do
11:                 N.distance(u, C) := GraphDist(G_0, u, C)
12:             for u ∈ U_1 do
13:                 N.distance(u, C) := GraphDist(G_1, u, C)
14:             for u ∈ S do                            ▷ separator vertices have zero distance to all components
15:                 N.distance(u, C) := 0
16:         N.left := NestedDissection(G_0, γ, δ)       ▷ dissect the left subgraph of G
17:         N.right := NestedDissection(G_1, γ, δ)      ▷ dissect the right subgraph of G
18:     else
19:         N.component := ∅                            ▷ mark leaf nodes of T by having no separator components
20:     return N
```

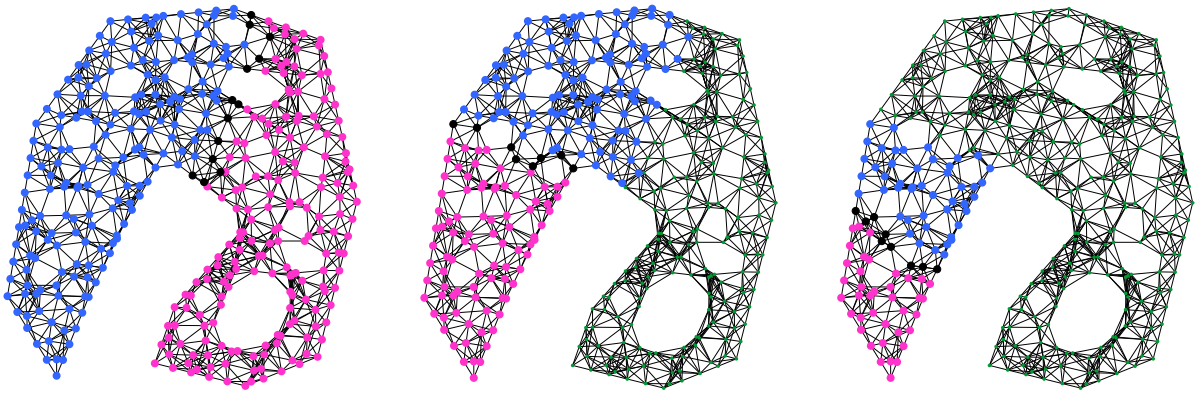

Figure 3: The nested dissection process: three levels of dissection for the unit disk graph UDG-SMALL3. Magenta and blue mark the vertices in each of the two components, and black vertices are the separator vertices at each level.

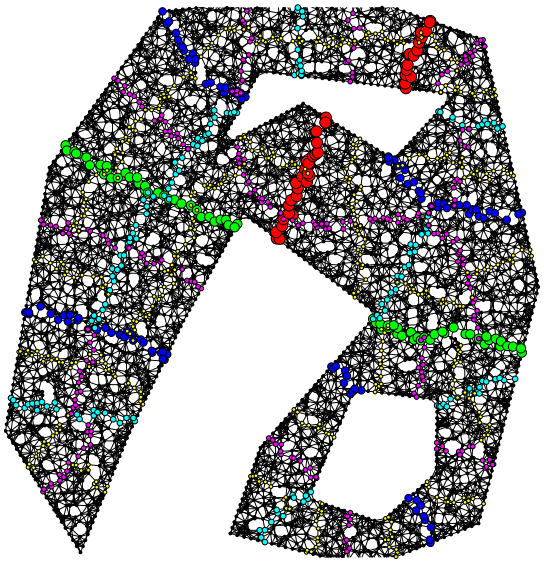

Figure 4: Complete set of vertex separators on the unit disk graph UDG-MEDIUM at 6 levels, colour-coded (red, green, blue, cyan, magenta, yellow) and size decreasing with depth. Zoom in to see the deeper separators.

induced by the separator. The leaf nodes of T store just the vertices of the corresponding cluster. SEPARATE(G) is the graph separation routine on G that returns two separate components $U_0, U_1 \subset V(G)$ and a vertex separator $S \subset V(G)$. PARTITION(S, γ) partitions a set of vertices S into components (i.e., subsets of vertices that are close to each other; see Figure 1) no larger than γ. GRAPHDIST(G, u, W) is a routine that computes the vector of shortest path distances in G between the vertex $u \in V(G)$ and the vertex subset $W \subset V(G)$. Figures 3 and 4 show examples of this nested dissection process.

Figure 5 illustrates (on plane graphs, for simplicity) the following observations that characterize the separator tree T and the separator subtrees T_v:

- Each vertex $v \in V$ is contained in at least one leaf of T (a cluster), but possibly in more than one. More precisely, the number of leaves containing v is equal to the number of times that v lies on a separator (plus one).
- T_v consists of all paths in T from the root to the leaves containing v (without the leaves themselves). In most cases, these paths coincide close to the root, so that the upper part of T_v is a single path, which bifurcates at a node with a separator containing v. Hence, if v is contained in a high-level separator, then the path bifurcates close to the root.
- Given two distinct vertices $v, w \in V$, the separator subtree $T_v \cap T_w$ is the union of the paths from the root to the least common ancestor (LCA) of any two leaves in T, one containing v and one containing w.
- Each subtree of a node containing a strong separator contains only v or w exclusively.

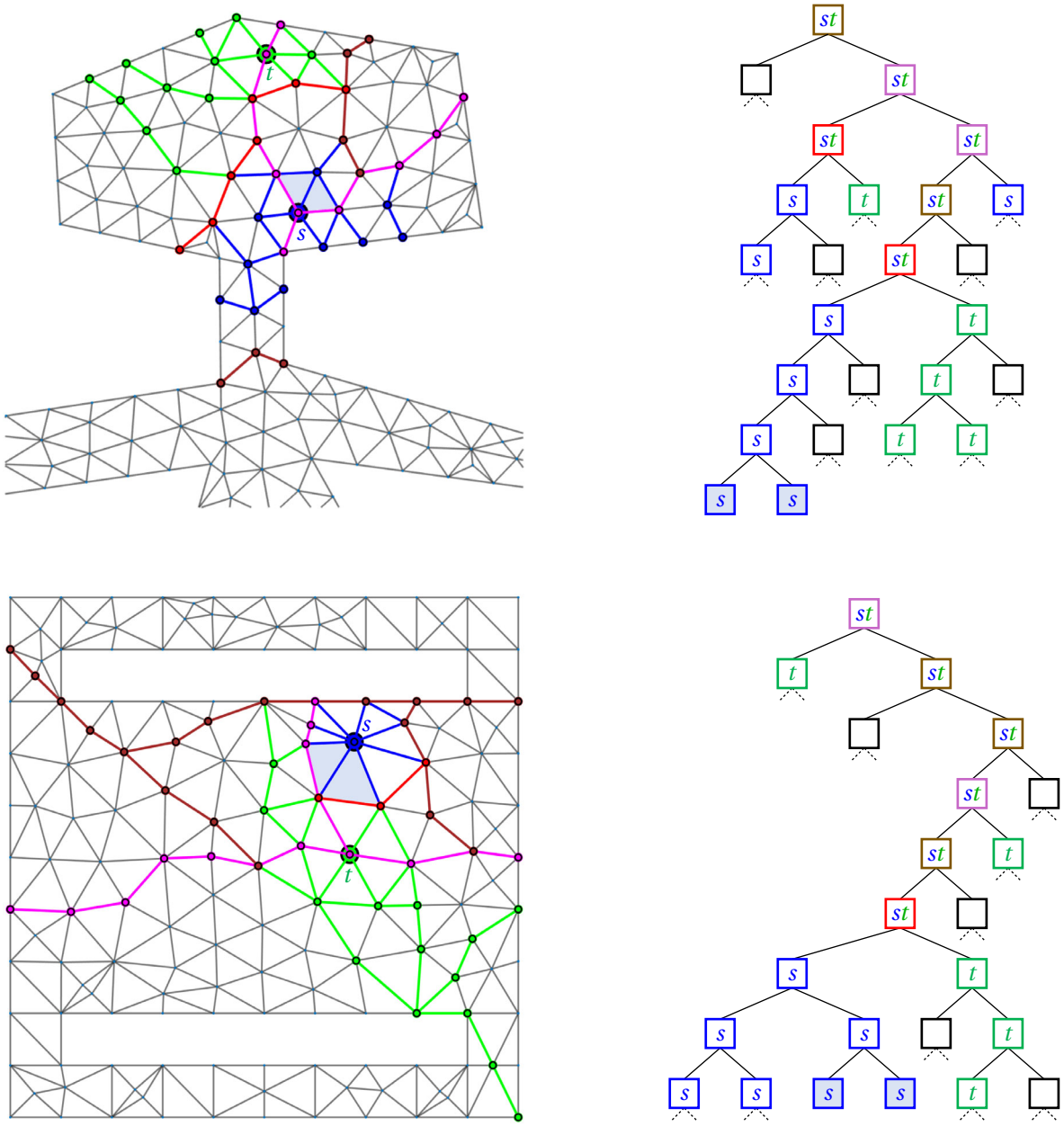

Figure 5: Two vertices, s (blue) and t (green), in two graphs (left) and their separators in the separator tree T (right). We use plane graphs in both examples. This allows for less cluttered figures, but the same observations and principles apply to UDGs. The root of T is at depth 0. Note that all the separators happen to be edge paths in the graph and the dissection is done down to the level of (diameter 1) cliques which are single edges or triangles. Each interior tree node represents a separator and is labelled with s and t depending on whether either of the two vertices is associated with this separator. Each node is coloured by the type of separator. Blue/green are those separators associated with only the blue/green vertex, respectively. Brown are those associated with *both* vertices but do not separate between them (at depths $0,3$ on the top and at depths $1,2,4$ on the bottom). Magenta are those associated with *both* vertices *and* separate them (at depths $1,2$ on the top and at depths $0,3$ on the bottom), and red are *strong* separators of s and t (at depth $2,4$ on the top and at depth 5 on the bottom). To illustrate, two of the leaves corresponding to the two blue shaded triangles containing the blue vertex are also shaded in blue.

2.4 Consequences of the random UDG model

If we use compact and balanced separators in Algorithm 1 and apply it to a random UDG (see Section 1.1), then we can rely on some bounds on the sizes of various elements of the UDG and the separator tree, as follows.

Theorem 1. *Let G be a random UDG with n vertices and edge density $\alpha = O(\log n)$ above the connectivity threshold α_C. If G is recursively bisected by balanced separators down to the level of clusters with diameter at most $\delta = O(1)$ and the separators are partitioned into components with at most $\gamma = \Omega(\sqrt{n})$ vertices, then the following bounds hold in expectation:*

1. *The number of vertices per cluster is $O(\log n)$.*
2. *The depth of the separator tree T is $O(\log n)$.*
3. *The number of vertices per separator is $O(1)$.*
4. *The number of clusters that contain a vertex v is $O(1)$.*
5. *The size of T_v is $O(\log n)$.*
6. *The number of separator components on T_v is $O(\log n)$.*

Proof. (1) Since the maximal diameter of each cluster is δ and the maximal edge length is 1, the maximal (physical) area covered by a cluster is $A = \pi(\delta/2)^2$, and so there are at most ρA vertices in each cluster. As the vertex density ρ is on the same order as the edge density α, it follows that the number of vertices per cluster is $O(\log n)$.

(2) Recall that the vertex separators are balanced and split a graph with n vertices into two components with at most $2n/3$ vertices. Adding the $O(\sqrt{n}\log n)$ separator vertices then results in two subgraphs with at most $2n/3 + \tau(n)$ vertices, where $\tau(n) = c\sqrt{n}\log n$ and c is some constant. Therefore, the depth $D(n)$ of a separator tree with n vertices satisfies the recurrence $D(n) \le D(2n/3 + \tau(n)) + 1$, and since $\tau(n) \le n/6$ for n sufficiently large, it follows that $D(n) \le D(5n/6) + 1$, hence $D(n) = O(\log n)$.

(3) Similarly, the total size $A(n)$ of all separators (i.e., the overall number of vertices in these separators) in a separator tree with n vertices satisfies the recurrence

$$A(n) \le A((1-\lambda)n + \tau(n)) + A(\lambda n + \tau(n)) + \tau(n), \qquad \lambda \in [1/3, 2/3].$$

By the Akra–Bazzi theorem [2], this implies $A(n) = O(n)$, and since the number of separators in the tree is $\Omega(n)$, the average number of vertices per separator is $O(1)$.

(4) It further follows that the average number of separators that contain a specific vertex v is $O(1)$, hence the number of clusters (or leaves in T) that contain v is $O(1)$, too.

(5) The average size of T_v is no larger than the depth of the tree multiplied by the average number of leaves (or clusters) containing v and therefore $O(\log n)$.

(6) The number $M(n)$ of separator components along a path from the root of a separator tree with n vertices to a leaf that contains v satisfies the recurrence $M(n) \le M(2n/3 + \tau(n)) + \lceil \tau(n)/\gamma \rceil$. Therefore, $M(n) = O(\sqrt{n}\log n/\gamma + \log n)$, and since $\gamma = \Omega(\sqrt{n})$, we conclude $M(n) = O(\log n)$. As the number of leaves that contain v is $O(1)$ by (4), the number of separator components on T_v is also $O(\log n)$. □

2.5 The basic routing scheme

Once the graph G has been recursively decomposed into a tree T, it is easy to see how routing from s to a given target vertex t may be achieved. By definition, if s and t are separated by vertex separator S, then any path between s and t must pass through S at some point. Thus a promising routing procedure, as illustrated in Figure 6, is to repeatedly move from s towards S. Once S is crossed at vertex σ, this principle repeats with a new separator deeper than S in T, separating between σ and t. This again repeats until the current vertex s is not separated at all from t, implying that it is contained in some cluster of the nested dissection as t is. From this point on, routing proceeds by standard routing tree methods within the small cluster. This routing algorithm is described in Algorithm 2. If S consists of just one component, then it is not hard to show that this algorithm always routes along the shortest path from s to σ, which is the closest point on the deepest separator S between s and t, and that this process repeats with s replaced by σ until a cluster containing t is reached. More precisely, S is always the "leftmost" deepest separator, if there happens to be more than one such separator. In the case that S is partitioned into multiple components, the algorithm routes to the component C of S with the smallest sum of distances to s and t.

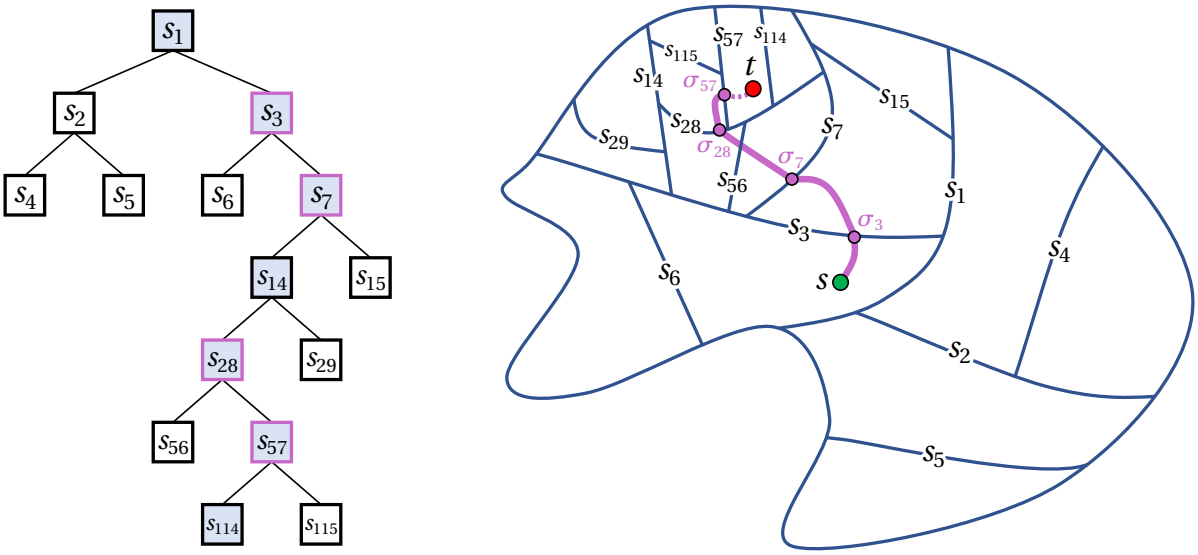

Figure 6: The basic routing scheme: The binary tree T (left) is part of the full binary tree representing the nested dissection of the graph (right). The separators S_i (not partitioned into components in this example) are labelled in the standard binary tree convention, where $2i$ and $2i+1$ are the labels of the descendants of node i. Blue nodes from root to leaf in T are those associated with target vertex t. The node containing S_3 is the LCA of the nodes associated with s and t, thus the first separator used in the routing process. The vertices σ_i are the crossing points of relevant S_i when routing to t. Purple bordered nodes in T are those which strongly separate s and σ_i from t, thus are used to route from s to t.

Algorithm 2 Basic routing from source s to target t in graph represented by tree node N

1: **procedure** RouteTree(N, s, t)
2: **while** N.component $\neq \emptyset$ **do** ▷ as long as N is not a leaf node of T,
3: **if** $s \in V_0(N)$ & $t \in V_0(N)$ **then** ▷ descend to the left subgraph, if it contains s and t,
4: $N := N$.left
5: **else if** $s \in V_1(N)$ & $t \in V_1(N)$ **then** ▷ descend to the right subgraph, if it contains s and t, or
6: $N := N$.right
7: **else** ▷ if s and t are strongly separated by S, then hop towards closest component of S
8: $(x, C) := \operatorname{argmin}_{\{(x,C):\, x \in \mathrm{Neigh}_{G(N)}(s),\, C \in N.\mathrm{component}\}} \big(1 + N.\mathrm{distance}(x, C) + N.\mathrm{distance}(t, C)\big)$
9: $s := x$
10: RouteCluster(N, s, t) ▷ route from s to t within the cluster represented by leaf node N

3 Compact routing

While the basic routing method described in Algorithm 2 is quite straightforward, the devil is in the details. The main challenge is to implement this method through independent (compact) routing tables at each graph vertex, and provide algorithm parameters, which control the storage-stretch tradeoff. This is non-trivial, as care must be taken that the routing does not get confused when a separator is approached, especially if it is "thick" and must be crossed without the route ending up in a loop (e.g., a loop of length one, i.e., a "sink"). Algorithm 2 achieves this by making routing decisions based on a "memory" of the node N in T that contains the current target separator $S(N)$. While this information could be stored and updated in a modifiable header during the routing, we would like to avoid that.

In this section we show how to circumvent this requirement by being a little more careful in choosing the separator to route to. This makes the algorithm just a little more complicated but has the additional benefit of achieving better stretch than the basic algorithm. Moreover, we note that the routing decision in each node requires access to the distances between t and the components of the separators associated with t (cf. line 8 of Algorithm 2). We describe how to retrieve this information in an initial handshaking step and store it in the packet header.

3.1 Distance estimates

Given a graph $G=(V,E)$, the length of a path of edges is defined as the number of those edges, and the distance $d_G(s,t)$ between two vertices $s,t\in V$ is defined as the length (i.e., hop counts) of the shortest path between s and t *within* G. This notion of distance between vertices may be extended to the distance between two *sets* of vertices $\mathcal{S}$ and $\mathcal{T}$ as

$$d_G(\mathcal{S},\mathcal{T})=\min_{s\in\mathcal{S},\,t\in\mathcal{T}} d_G(s,t).$$

Note that if G' is a subgraph of G, then

$$d_G(s,t)\le d_{G'}(s,t)$$

for all $s,t\in V(G')$, because it may happen that the shortest path between s and t in G is not entirely in G'. We further denote the neighbours of s within G' by

$$\mathrm{Neigh}_{G'}(s)=\{u\in V(G'):(s,u)\in E\}.$$

Recall that the key quantity in our routing method, stored in the separator tree, is the distance between a vertex v and a separator component C. Note that for two vertices $s,t\in V(G)$, if they are separated by S, partitioned into certain C, in two connected components G_0 and G_1, then a reasonable approximation of the distance $d_G(s,t)$ is the *estimated distance through* S,

$$e_G^S(s,t)=\min_{C\subseteq S}\{d_{G_0}(s,C)+d_{G_1}(t,C)\},\tag{1}$$

where we assume without loss of generality that $s\in V_0(S)$ and $t\in V_1(S)$. More precisely, it follows from the triangle inequality that the estimated distance is a lower bound on the true distance. Indeed, if $u\in S$ is any vertex on the shortest path from s to t and C' is the separator component that contains u, then

$$e_G^S(s,t)\le d_{G_0}(s,C')+d_{G_1}(t,C')\le d_{G_0}(s,u)+d_{G_1}(t,u)=d_G(s,t).$$

As we shall see later, these distance estimates, in particular from the target t, will guide the routing towards t.

Since the separators generated by the nested dissection are in one-to-one correspondence with the internal nodes of the tree T, specifying a separator S uniquely determines the node $N(S)$ that contains it, thus also the subgraph $G(N)$ that it separates into $G_0(N)$ and $G_1(N)$. As such, we may omit the subscript G and use the notation e_S instead of e_G^S.

3.2 The routing tables

The first step in the routing method is the offline construction of a *routing table* for each graph vertex v. Each routing table should contain enough information to determine which of v's neighbours to proceed to, given a target vertex t, and should be such that a packet is always guaranteed to eventually arrive at t without getting stuck in an infinite loop of any sort. The routing table of v consists of two parts, the *cluster routing table* $\mathsf{CRT}(v)$ and (a description of) the *separator routing tree* $\mathsf{SRT}(v)$, generated by Algorithm 3.

CRT is a standard list of pairs (t_i,u_i) which indicates that if the target vertex is t_i, then the routing should proceed to neighbour u_i of v. The vertex u_i is just the predecessor of v in the shortest path tree rooted at t_i within the cluster that contains v and t_i. In principle, it is possible to build such a table for all possible targets in the graph, but then the size of the table would be a prohibitive $O(n)$. So in practice, this table contains routing information in CRT only for the small number of vertices which happen to be in a cluster which also contains v. Moreover, we pay attention to store the neighbour on the shortest of all shortest paths, in case v and t_i are in more than one common cluster. By definition of the clusters, the distance of these vertices from t is at most δ.

For all other vertices t more distant from v, there is no explicit information in the cluster routing table on how to get closer to t. This is where the more sophisticated part of the routing table, the SRT, comes in. In this case, routing is in the direction of a separator S between v and t, building on the separation property, namely that any path between v and t *must* intersect S, or rather one of its components at some point. Thus we must ensure that all other target vertices have at least one separator between them and v and information about the components C of this separator is present in v's SRT. More precisely, SRT contains for each separator component C associated with some neighbour of v, the length of the shortest path from v to C passing through a neighbour u of v associated with C, as well as u itself. To store this information, $\mathsf{SRT}(v)$ needs to have the same structure as the union $\mathcal{T}_v=\bigcup_{u\in\mathrm{Neigh}_G(v)}T_u$ of the separator subtrees of v's neighbours.

At this point, we can show that the two components of the routing tables are indeed compact.

Algorithm 3 Construct the routing table parts CRT and SRT and the routing header RH for vertex v of $G(V,E)$ from T

function BUILDROUTINGTABLE(G, T, v)
  1) build CRT
  $\mathcal{N}_v := \{N \in \text{leaves}(T) : v \in N.\text{vertices}\}$ ▷ find all leaf nodes of T representing clusters containing v
  $L := \infty$ ▷ initialize a vector of infinite values
  for $N \in \mathcal{N}_v$ **do**
    for $t \in N.\text{vertices} \setminus \{v\}$ **do** ▷ for all vertices that share some cluster with v,
      $(\Gamma, l) := \text{SHORTESTPATH}(G(N), v, t)$ ▷ find shortest path from v to t and its length
      if $l < L(t)$ **then** ▷ check, if it is shorter than any previous shortest path to t
        $\mathsf{CRT}(t) := \Gamma(2)$ ▷ second vertex along this path is best neighbour of v for target t
        $L(t) := l$
  2) build SRT
  $\mathcal{T}_v := \bigcup_{u \in \text{Neigh}_G(v)} T_u$ ▷ merge separator subtrees of v's neighbours
  create SRT with the same structure as $\mathcal{T}_v$
  for $N \in \text{nodes}(\mathcal{T}_v)$ **do** ▷ for all separators associated with some neighbour of v and
    let N' be the node in SRT corresponding to N
    for $C \in N.\text{component}$ **do** ▷ for any component C of these separators, find
      $u := \underset{\{u \in \text{Neigh}_{G(N)}(v)\}}{\arg\min} \big(1 + N.\text{distance}(u, C)\big)$ ▷ neighbour on shortest path from v to C
      $N'.\text{next}(C) := u$ ▷ remember this neighbour
      $N'.\text{nextDistance}(C) := 1 + N.\text{distance}(u, C)$ ▷ and the length of this shortest path
  3) build RH
  create RH with the same structure as T_v
  for $N \in \text{nodes}(T_v)$ **do**
    let N' be the node in RH corresponding to N
    for $C \in N.\text{component}$ **do** ▷ for any component C of a separator associated with v,
      $N'.\text{distance}(C) := N.\text{distance}(v, C)$ ▷ remember the distance to C
  return $(\mathsf{CRT}, \mathsf{SRT}, \mathsf{RH})$

Corollary 2. *With the assumptions of Theorem 1, the average size of* $\mathsf{CRT}(v)$ *is* $O(\log n)$ *and the average size of* $\mathsf{SRT}(v)$ *is* $O(\log^2 n)$.

Proof. The bound on the average size of $\mathsf{CRT}(v)$ is an immediate consequence of property (1) in Theorem 1. Moreover, as the average number of neighbours is 2α with $\alpha = O(\log n)$, it follows from property (5) of Theorem 1 that the average size of $\mathcal{T}_v$ is $O(\log^2 n)$ and further from property (6) of Theorem 1 that the average size of $\mathsf{SRT}(v)$ is also $O(\log^2 n)$. □

3.3 The routing header

Given a target vertex t, our routing algorithm relies on a header that characterizes t and provides all the information required throughout the routing procedure to guide the routing towards t. In essence, these are the distances of t to all the components of its associated separators, which are contained in the separator subtree T_t and copied into the structure $\mathsf{RH}(t)$ (see Algorithm 3). So the size of $\mathsf{RH}(t)$ is $O(\log n)$ with a very small constant (just three integers for each separator component associated with t). At the beginning of the routing procedure, this *routing header* $\mathsf{RH}(t)$ is appended to the packet. To this end, we could store each routing header in the corresponding network node and implement a handshaking mechanism by which s retrieves $\mathsf{RH}(t)$ from t. Alternatively, we could store *all* routing headers in a small number of "hub" nodes and in each node the routing header of the nearest hub, so that our routing algorithm itself can be used for retrieving $\mathsf{RH}(t)$, similar to a DNS lookup in TCP/IP. Other variants are possible and may depend on the particular application, but in any case the resulting overhead is expected to be small in comparison to the cost of routing a long stream of packets. Hence, we excluded the handshake in our implementation and the results reported below.

Algorithm 4 Route from source s to target t using the routing tables (CRT and SRT) and the routing header $\mathsf{RH}(t)$

1: **procedure** ROUTE(s, t, h)
2: **while** $s \neq t$ **do** ▷ route until the target is reached
3: **if** $i = \text{find}(t, \text{col}_1(\mathsf{CRT}(s)))$ **then** ▷ if t is among the targets in $\mathsf{CRT}(s)$, i.e., in some common
4: $s := u_i$ ▷ cluster with s, then use $\mathsf{CRT}(s)$ to get closer to t
5: **else** ▷ otherwise, use $\mathsf{SRT}(s)$ to route towards a deepest common separator between s and t
6: $\mathcal{N} := \{N \in \mathsf{SRT}(s) : N$ is a deepest node having a common separator with some corresponding node $N' \in \mathsf{RH}(t)\}$
7: $(N, C) := \underset{\{(N,C): N \in \mathcal{N}, C \subseteq S(N)\}}{\arg\min} \big(N.\text{nextDistance}(C) + N'.\text{distance}(C)\big)$
8: $s := N.\text{next}(C)$

3.4 The routing algorithm

The routing of a packet from s to t proceeds by using the routing table at s to determine which of the neighbours of s to forward the header and packet to, as described in Algorithm 4. As we will see later, Algorithm 4 is more sophisticated than the basic Algorithm 2. The former, by definition, always proceeds *down* through the tree, whereas the latter also allows *lateral* moves in the tree, if that brings the packet closer to the target. After all, this is not so surprising, because without the "memory" of the current separator, the algorithm cannot simply "descend" in the underlying separator tree, but rather has to consider all possible paths in the tree that lead to a cluster containing t.

Before we start, let us establish a simple, but key property of the separator tree (cf. Figure 5).

Lemma 3. *Let $u, v \in V$ be two distinct vertices, N a deepest node of $T_u \cap T_v$, and S the separator at N. Then, u and v are either strongly separated by S or in a common cluster.*

Proof. If u and v are not strongly separated by S, then we can assume without loss of generality that $u, v \in V_0(N)$. But as N is a deepest node among all nodes containing both u and v, this is possible only if $G_0(N)$ is a cluster. □

For a fixed target vertex t and any vertex $u \in V$ that is not in a common cluster with t, let k_u be the depth of the deepest nodes in $T_u \cap T_t$ and $\mathcal{S}_u$ be the set of *deepest* separators between u and t corresponding to these deepest nodes. Note that deepest separators are necessarily strong separators by Lemma 3. We denote the smallest distance estimate (1) based on these deepest separators by $e_u = \min_{S \in \mathcal{S}_u} e_S(u,t)$, and let $S_u = \arg\min_{S \in \mathcal{S}_u} e_S(u,t)$ be a corresponding separator. Further let N_u be the node associated with S_u and C_u be a component of S_u with $e_u = d_{G_0}(u, C_u) + d_{G_1}(t, C_u)$, where $G_0 = G_0(N_u)$ and $G_1 = G_1(N_u)$, assuming without loss of generality that $u \in V(G_0)$ and $t \in V(G_1)$. In other words, among all estimated distances between s and t through one of the components of the separators in $\mathcal{S}_u$, the one through C_u is the smallest.

With these definitions, we are now ready to prove that the separator routing trees $\mathsf{SRT}(v)$ can be used to repeatedly bring the packet closer to a cluster that contains t, either by descending a level in T or by reducing the remaining estimated distance to t. Once such a cluster is reached, the final delivery to t is handled by the cluster routing tables $\mathsf{CRT}(v)$.

Theorem 4. *For any vertex s, let v be the neighbour of s to which the* ROUTE *procedure moves in line 8 of Algorithm 4. Then, either $k_v > k_s$ or $k_v = k_s$ and $e_v \leq e_s - 1$.*

Proof. Let S_s be the deepest separator in $T_s \cap T_t$ as defined above and N_s be the node associated with S_s. Since we only get to line 8 in Algorithm 4 as long as s is not in a common cluster with t, we conclude from Lemma 3 that S_s is a strong separator of s and t, and can hence assume without loss of generality that $s \in U_0(N_s)$ and $t \in U_1(N_s)$. Let r be the closest vertex to s in C_s, that is, $d_{G_0}(s,r) = d_{G_0}(s, C_s)$, where $G_0 = G_0(N_s)$ and $G_1 = G_1(N_s)$. Further let Γ be a shortest path from s to r within G_0, and $u \in \text{Neigh}_{G_0}(s)$ be the neighbour of s along Γ (see Figure 7).

We first observe that

$$e_s = d_{G_0}(s, C_s) + d_{G_1}(t, C_s) = d_{G_0}(s,r) + d_{G_1}(t, C_s), \tag{2}$$

by the definition of r, and

$$d_{G_0}(u, C_s) = d_{G_0}(u,r) = d_{G_0}(s,r) - 1, \tag{3}$$

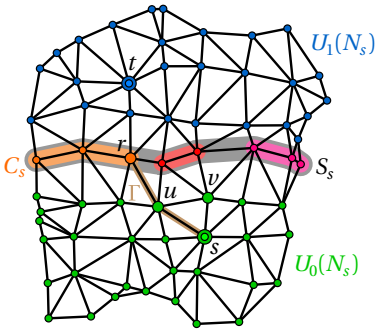

Figure 7: Notation used in the proof of Theorem 4.

.

by the definition of u and the fact that the path from u to r along Γ is a shortest path, too. Moreover,

$$k_u \leq k_v, \tag{4}$$

because k_v is the depth of a deepest node in the tree $\mathcal{T}_s \cap T_t$, which contains $T_u \cap T_t$ by definition.

Since $s \in U_0(N_s)$ implies $u \in V_0(N_s) = U_0(N_s) \cup S_s$, we know that S_s is also a (possibly weak) separator of u and t corresponding to some node in $T_u \cap T_t$, which is not necessarily among the deepest nodes, hence $k_s \leq k_u$. If $k_s < k_u$, then (4) implies that the theorem holds, and likewise in the case that $k_u < k_v$. It remains to consider the case $k_s = k_u = k_v$. In this case, S_s is among the deepest separators in $\mathcal{S}_u$ and

$$e_u = \min_{S \in \mathcal{S}_u} e_S(u,t) \leq e_{S_s}(u,t) \leq d_{G_0}(u, C_s) + d_{G_1}(t, C_s) = e_s - 1, \tag{5}$$

where the last equality follows from (2) and (3). Our goal now is to show that the estimated distance $1 + e_u$ from s to t via u cannot be smaller than the estimated distance $1 + e_v$ via v, since that would contradict the choice of v as the best neighbour to route to. To this end, recall that e_v is the smallest estimated distance from v to t through any deepest separator in $\mathcal{S}_v$, which are among the deepest separators in $\mathcal{T}_s \cap T_t$. Therefore, by the definition of N.nextDistance in line 19 of Algorithm 3,

$$1 + e_v \leq 1 + d_{G_0(N)}(v, C) + d_{G_1(N)}(t, C) = N.\text{nextDistance}(C) + d_{G_1(N)}(t, C),$$

where N and C are the optimal deepest node and component selected in line 7 of Algorithm 4. Since also the separators in $\mathcal{S}(u)$ are among the deepest separators in $\mathcal{T}_s \cap T_t$, it follows from the definition of N, C, N.nextDistance, and e_u that

$$\begin{aligned} N.\text{nextDistance}(C) + d_{G_1(N)}(t, C) &\leq N_u.\text{nextDistance}(C_u) + d_{G_1(N_u)}(t, C_u) \\ &\leq 1 + d_{G_0(N_u)}(u, C_u) + d_{G_1(N_u)}(t, C_u) \\ &= 1 + e_u. \end{aligned}$$

Overall, this gives the desired inequality

$$1 + e_v \leq 1 + e_u \leq e_s,$$

with the second inequality following from (5). □

Note that the most important step in Algorithm 4, which is also key for the proof of Theorem 4, is selecting the *deepest* separator. To illustrate this, consider the following simple scenario where not choosing the deepest separator results in the router entering a loop:

Suppose that $(s, r) \in E$ and that s, r, and t all lie on some separator S represented by node N, but s and t are not in a common cluster. Further assume that during the construction of $\mathsf{SRT}(s)$, r is chosen in line 17 of Algorithm 3 for some component $C \subseteq S$, so that $N.\text{next}(C) = r$ in $\mathsf{SRT}(s)$, and vice versa during the construction of $\mathsf{SRT}(r)$ for the same component C. As $N'.\text{distance}(C) = 0$ in $\mathsf{RH}(t)$, it could happen that ROUTE chooses (N, C) in line 7 of Algorithm 4 and hence r as the next vertex to hop to, if $\mathcal{N}$ was simply the set of common separators between s and t. Alas, once we are at r, symmetry implies that we could route back to s, and a loop is formed. The problem in this scenario is that S is not a strong separator of s and t.

However, as ROUTE considers only deepest common separators in $\mathcal{N}$ (cf. line 6), which strongly separate s and t by Lemma 3, then the symmetry is broken and the routing procedure eventually reaches a cluster that contains t, as stated by Theorem 4.

We now prove the central property of our routing scheme, namely, that Algorithm 4 delivers.

Corollary 5. *The procedure* ROUTE *terminates at* t.

Proof. We first observe that the else-branch in the while loop is executed only a finite number of times. This follows directly from Theorem 4, by noticing that the pair $(-k_s, e_s)$ decreases strictly monotonically with respect to lexicographic order in each step of the while loop. Hence, no vertex can be visited twice, and since the number of vertices is finite, this part of the procedure terminates when s enters a cluster containing t.

From this point on, the definition of the cluster routing tables (cf. Algorithm 3) guarantees that we route along a path towards t that is at most as long as the shortest of all shortest paths from s to t within the clusters that contain s and t. Indeed, if N is the leaf node of T that represents the cluster $V(N)$ containing this shortest path Γ from s to t and u is the neighbour of s to which the ROUTE procedure moves in line 4 of Algorithm 4, then $u \in V(N)$ and the shortest path from u to t in the cluster $V(N)$ is the one following Γ. Hence, $\mathsf{CRT}(u)$ will continue to route along Γ, unless there is an even shorter path from u to t in some other cluster. □

4 Experimental results

We have implemented our routing method in MATLAB and run it on a number of sample inputs, consisting of UDGs of varying size n and edge density α. Table 1 provides the specifications of these graphs, and they are shown in Figure 8. Although the standard random UDG model features vertices scattered uniformly in a square, we make it a little more interesting and realistic (and challenging) by using a non-simply-connected polygonal region instead of a square. The first set of UDGs (UDG-SMALL) were generated by scattering $n = 379$ points randomly within a non-convex polygonal region with two "holes", bounded by the unit square, and then connecting with an edge any two vertices whose distance was less than a parameter, which took values in $\{0.05, 0.06, 0.07, 0.08, 0.09\}$, resulting in graphs with identical vertex sets, but with $m = 876$, 1222, 1650, 2181, and 2809 edges, respectively, corresponding to edge density α values between 2.07 and 7.41. The second UDG (UDG-MEDIUM) contained $n = 4563$ vertices in the same polygonal region with max edge length 0.0175, resulting in $m = 19694$ edges ($\alpha = 4.32$). The third (UDG-LARGE) contained $n = 26542$ vertices in the same polygonal region with max edge length 0.007, resulting in $m = 113924$ edges ($\alpha = 4.29$). The connectivity threshold of $\alpha_C \approx 0.5 \log n$ translates to 2.96, 4.21 and 5.09 for UDG's of these sizes respectively, close to the edge density α of UDG-SMALL3, UDG-MEDIUM and UDG-LARGE, thus we show our results on these graphs.

Balanced vertex separators for the nested dissection of the graphs were computed using a simple spectral partitioner, implemented in just 15 lines of MATLAB code. Nested dissection was done up to a cluster with diameter parameter δ ranging between 1 and 5. Separators were partitioned into components of parameter size γ, where $\gamma = 1, 2, 4, \ldots$, up to half the maximal separator size, as well as not partitioned at all by setting γ to the maximal separator size. Note that the condition $\gamma = \Omega(\sqrt{n})$, which guarantees routing tables of size $O(\log^2 n)$ by Corollary 2, is covered by the last two cases.

For each of the inputs and algorithm parameters, we performed nested dissection, as described in Algorithm 1, and constructed the routing tables CRT and SRT for all graph vertices, as described in Algorithm 3. We then measured the following: (1) average cluster depth in the separator tree, (2) average size (number of nodes) of $\mathsf{CRT}(v)$, (3) total number of separator components, (4) average size (entries) in $\mathsf{SRT}(v)$, (5) average, (6) standard deviation, and (7) maximal stretch of the resulting routes. For the small graphs (UDG-SMALL), we computed (5), (6), and (7) by running our routing algorithm ROUTE (Algorithm 4) exhaustively over all possible (s, t) input pairs of vertices. For the larger graphs (UDG-MEDIUM and UDG-LARGE), this was done on a random sample of 1000 vertex pairs. These measurements are reported in Tables 2–4.

The cluster diameter parameter δ determines the termination criterion for the recursive nested dissection. The value 1 means that clusters are cliques. In general, the depth of the separator tree decreases with an increase in this value. This parameter also determines the average size of the CRT part of the routing table per vertex, which will increase with δ. Partitioning separators into multiple components (as determined by the maximal component size parameter γ) will increase the total number of such components in the system, thus also the average size of the SRT part of the routing table per vertex. However, it will decrease the resulting stretch. So by adjusting the parameters, it is possible to tradeoff the average size of the routing

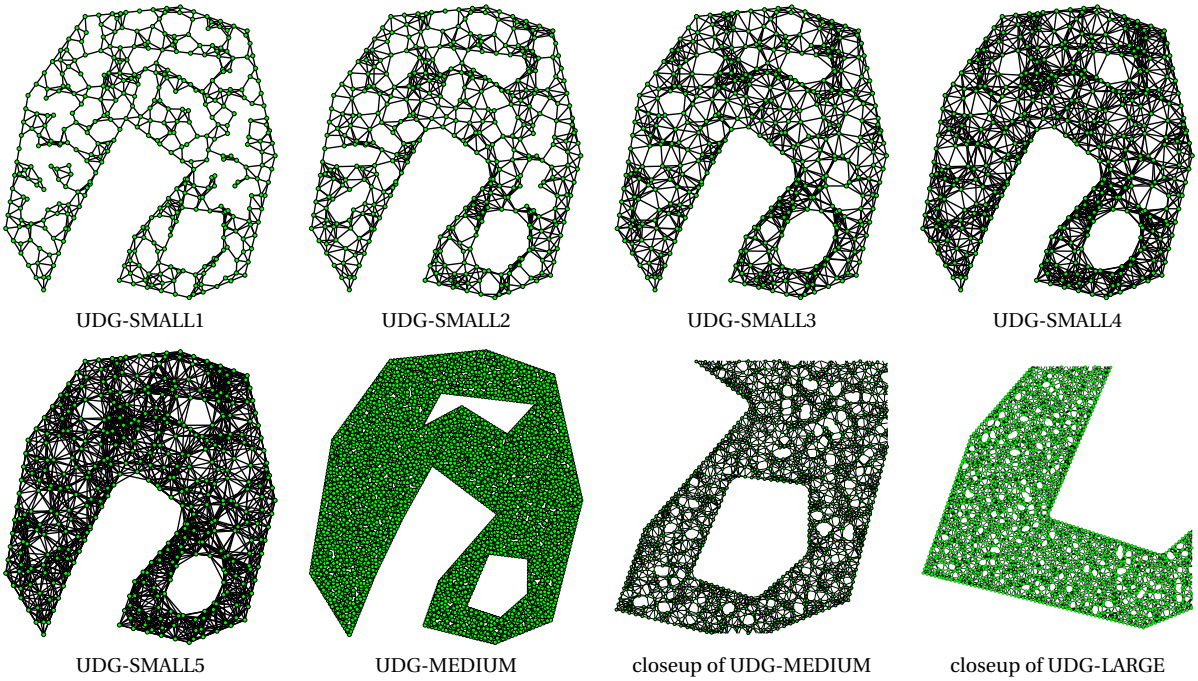

Figure 8: UDGs used in our experiments, whose specifications are in Table 1. All UDG-SMALL graphs have common vertex sets, but different edge densities.

Table 1: Graphs used in our experiments.

name	# vertices (n)	# edges (m)	edge density ($\alpha = m/n$)
UDG-SMALL1	379	876	2.07
UDG-SMALL2	379	1222	3.22
UDG-SMALL3	379	1650	4.35
UDG-SMALL4	379	2181	5.75
UDG-SMALL5	379	2809	7.41
UDG-MEDIUM	4563	19694	4.32
UDG-LARGE	26542	113924	4.29

table: $|\mathsf{CRT}|+|\mathsf{SRT}|$ for stretch. The best stretch is obtained when separators are partitioned to components of size $\gamma = 1$ (singletons), however, the cost in routing table size is prohibitive, namely $O(\sqrt{n}\log^2 n)$.

The charts in Figure 9 visualize the most important data columns from Tables 2–4. For these UDGs with increasing vertex size, but constant edge density of approximately $\alpha = 4.35$ edges/vertex (equivalent to an average of 8.7 neighbours per vertex), taking $\delta = 5$ and γ to be half of the maximal separator size seems to be a good combination. It is most interesting to see how the resulting average sizes of CRT and SRT scale with the number of vertices. We expect $|\mathsf{CRT}|$ to scale like $O(\log n)$ with n. Indeed, for $n = 4563$ we have $|\mathsf{CRT}| = 35.4$, and for $n = 26542$ we have $|\mathsf{CRT}| = 36.4$, whose ratio of 0.97 is quite close to the ratio $\log(4563)/\log(26542) = 0.83$. On the other hand, we expect $|\mathsf{SRT}|$ to scale like $O(\log^2 n)$ logarithmically with n. Indeed, for $n = 4563$ we have $|\mathsf{SRT}| = 17.7$ and for $n = 26542$ we have $|\mathsf{SRT}| = 20.9$, whose ratio of 0.85 is quite close to the ratio $\log^2(4563)/\log^2(26542) = 0.69$. The resulting average stretches are 1.09 and 1.11, respectively.

To compare our results with those of the S4 routing method of Mao et al. [24], we state the conclusion from their experiments that to achieve stretch between 1.05 and 1.10 with S4 requires the storage requirements to scale like $O(\sqrt{n})$, whereas the overall size of our routing tables is $O(\log^2 n)$, due to the fact that we use balanced separators which guarantee that the depth of the separator tree is $O(\log n)$.

Figure 10 compares some of the paths generated by our compact routing algorithm with the true shortest paths between pairs of vertices. Figure 11 compares the complete compact routing trees on some select vertices to the shortest-path routing trees for those same targets.

Table 2: Experimental results for UDG-SMALL3 (cf. Table 1). Parameters of the routing table are δ (the maximal cluster diameter in the nested dissection) and γ (the maximal size of the separator components). The notation $[x]$ indicates the average of quantity x. Stretch results are given as average ± standard deviation.

δ	[cluster depth]	[|CRT|]	γ	#components	[|SRT|]	[stretch]	max stretch
1	13.3	8.7	1	7132	514.8	1.03 ± 0.06	2.20
			2	4005	286.4	1.05 ± 0.07	2.28
			4	2379	169.0	1.06 ± 0.07	2.11
			17	1813	119.3	1.09 ± 0.11	2.21
2	7.6	14.1	1	828	119.3	1.03 ± 0.05	2.08
			2	459	64.6	1.04 ± 0.06	2.26
			4	270	37.5	1.05 ± 0.07	2.17
			17	167	18.9	1.09 ± 0.11	2.96
3	6.8	20.0	1	420	87.9	1.02 ± 0.05	2.21
			2	227	46.7	1.03 ± 0.06	2.36
			4	133	26.9	1.05 ± 0.07	2.53
			17	66	11.2	1.08 ± 0.11	2.46
4	6.1	25.2	1	278	73.2	1.02 ± 0.05	2.20
			2	147	38.5	1.03 ± 0.06	2.86
			4	85	21.9	1.05 ± 0.07	2.36
			17	39	8.5	1.08 ± 0.11	3.14
5	5.7	29.7	1	229	67.2	1.02 ± 0.05	2.23
			2	120	35.2	1.03 ± 0.06	1.99
			4	69	20.0	1.04 ± 0.06	2.25
			17	28	7.1	1.08 ± 0.11	2.49

Table 3: Experimental results for UDG-MEDIUM (cf. Table 1).

δ	[cluster depth]	[|CRT|]	γ	#components	[|SRT|]	[stretch]	max stretch
2	11.5	14.2	5	3231	68.7	1.06 ± 0.09	3.05
			10	2563	43.7	1.07 ± 0.10	2.30
			20	2469	33.5	1.09 ± 0.10	3.43
			42	2452	28.2	1.11 ± 0.13	3.05
3	10.2	20.1	5	1697	58.7	1.05 ± 0.09	2.92
			10	1148	34.5	1.07 ± 0.10	3.35
			20	1054	24.3	1.09 ± 0.10	3.60
			42	1037	19.0	1.11 ± 0.12	3.06
4	9.3	26.8	5	1125	53.6	1.05 ± 0.08	2.60
			10	684	30.5	1.07 ± 0.09	2.90
			20	591	20.3	1.09 ± 0.10	3.14
			42	574	15.0	1.11 ± 0.12	3.66
5	8.6	35.4	5	791	49.9	1.05 ± 0.08	2.92
			10	456	27.9	1.07 ± 0.10	2.97
			20	363	17.7	1.09 ± 0.10	3.76
			42	346	12.4	1.11 ± 0.12	3.37

Table 4: Experimental results for UDG-LARGE (cf. Table 1).

δ	[cluster depth]	[|CRT|]	γ	#components	[|SRT|]	[stretch]	max stretch
5	11.3	36.4	10	2873	59.2	1.05 ± 0.08	3.54
			20	2258	34.4	1.07 ± 0.09	4.00
			50	2133	20.9	1.11 ± 0.10	3.18
			102	2103	16.2	1.12 ± 0.12	3.02

We conclude by mentioning that even though the graph UDG-SMALL-1 has edge density $\alpha \leq 0.5 \log n$, we did not encounter any significant problems with maintaining connectivity throughout the nested dissection process and obtained results consistent with the ones reported above.

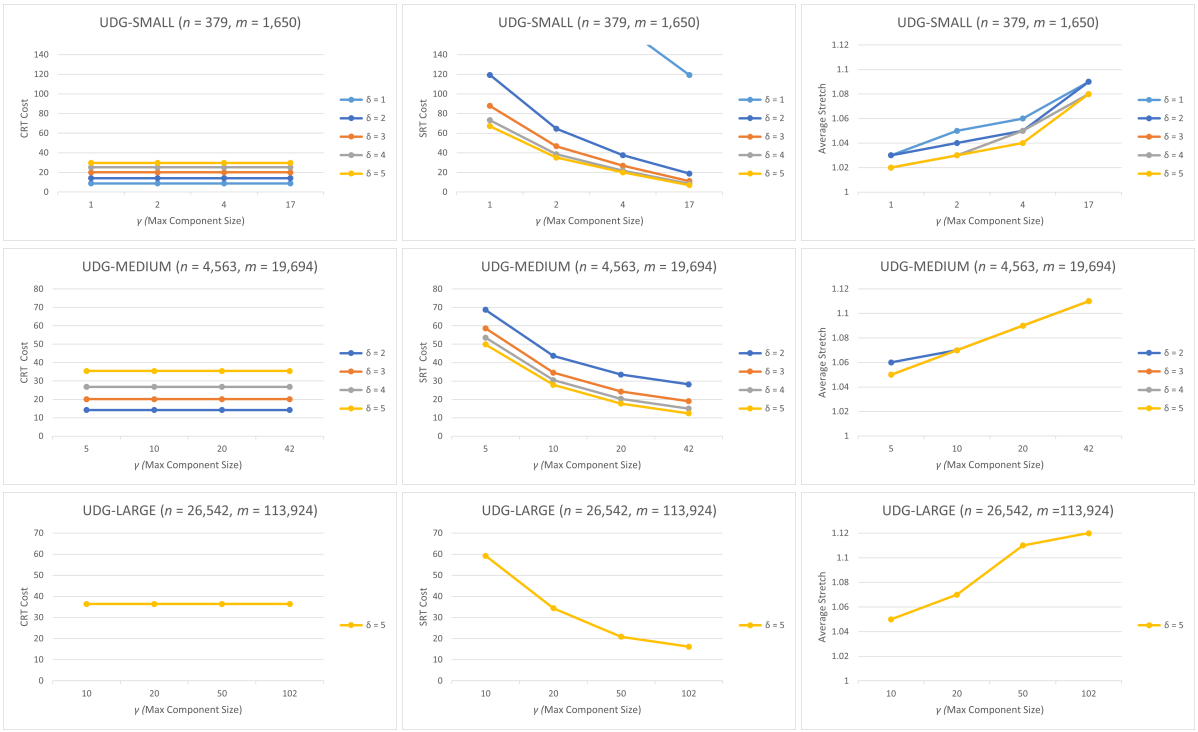

Figure 9: Average routing table CRT and SRT sizes and stretch for UDG-SMALL3, UDG-MEDIUM and UDG-LARGE (at edge density approximately 4.35) vs. max component size γ for different values of cluster diameter δ.

5 Discussion

We have assumed a theoretical model of a *connected* random UDG. This implies that the edge density α is $O(\log n)$, as will be the average number of neighbours per vertex. For this model, we have described a simple, yet effective, algorithm to construct compact routing tables for the UDGs, relying on the existence of compact balanced vertex separators. The efficiency of the resulting routing tables in terms of the state-stretch tradeoff depends on the quality of these separators. This, in turn, depends on the values of the algorithmic parameters δ (cluster diameter) and γ (vertices per separator component). Based on the theoretical complexities, which have been confirmed by our experimental results, the optimal values of these parameters are $\delta = O(1)$ (a small constant) and $\gamma = \Omega(\sqrt{n})$, in which case we get routing tables of size $O(\log^2 n)$.

In our implementation, we used simple methods which generate reasonable separators, but this may be further improved by using a more sophisticated algorithm to improve the quality of the separators. Moreover, we made no attempt to minimize the number of bits per routing table entry, and it is likely that this number can be reduced from $O(\log n)$ to $O(1)$ by utilizing a local indexing scheme rather than global vertex indices. Finally, we should point out that our method is not suitable for *dynamic* graphs, where the connectivity may vary over time, since any such change would require the routing tables to be recomputed.

Acknowledgements

We thank the anonymous reviewers for providing very helpful comments and suggestions.

References

[1] I. Abraham, C. Gavoille, and D. Malkhi. Compact routing for graphs excluding a fixed minor. In P. Fraigniaud, editor, *Distributed Computing*, volume 3724 of *Lecture Notes in Computer Science*, pages 442–456. Springer, Berlin, Heidelberg, 2005.

[2] M. Akra and L. Bazzi. On the solution of linear recurrence equations. *Computational Optimization and Applications*, 10(2):195–210, May 1998.

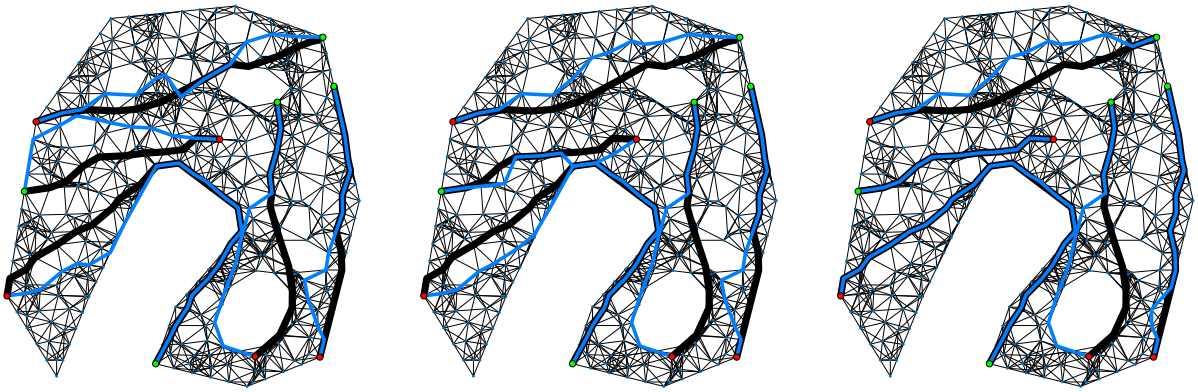

Figure 10: Sample (blue) paths generated by our compact routing algorithm compared to the true (black) shortest paths for some pairs of (red) s and (green) t vertices for the UDG-SMALL3 graph with parameters $\delta = 3$ and $\gamma = 17, 4, 1$ (left to right), reducing the stretch.

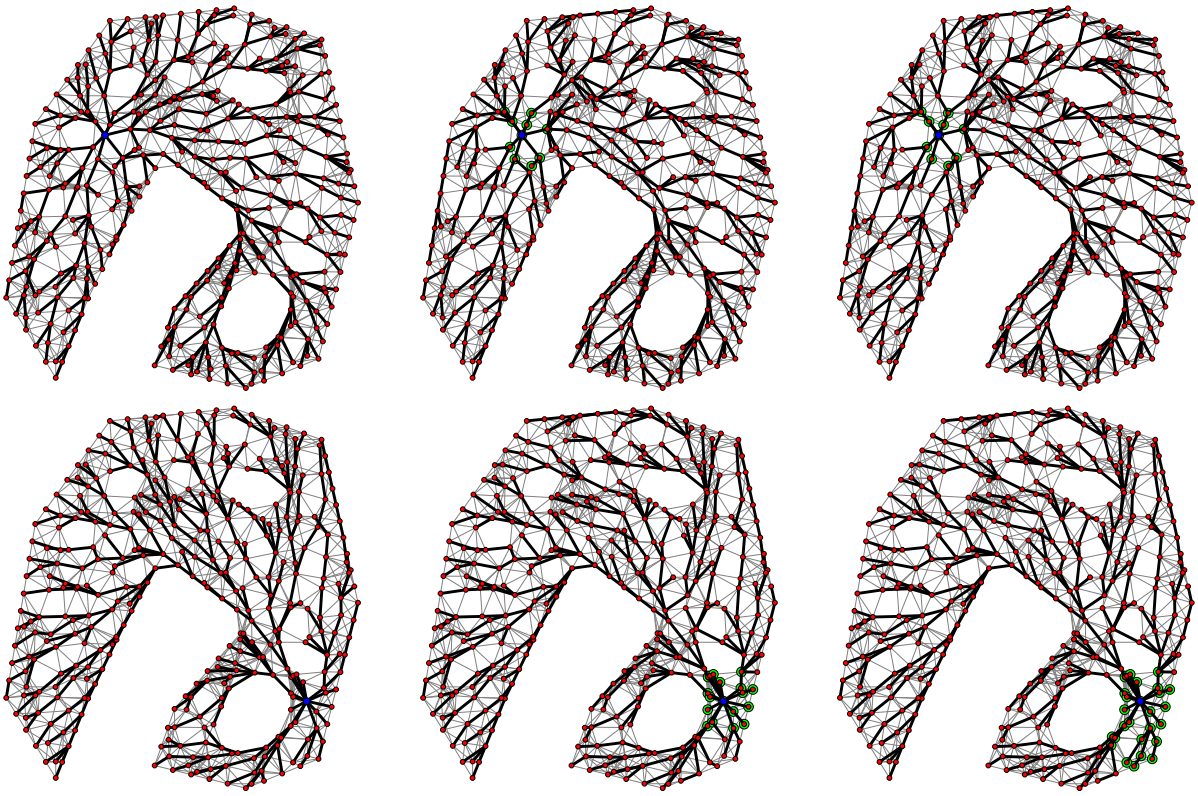

Figure 11: Comparison of routing trees for two arbitrary target vertices of the UDG-SMALL3 graph: shortest-path routing trees (left) versus routing trees generated by our compact routing algorithm for same target vertices with $\delta = 2$ (middle) and $\delta = 3$ (right), both with $\gamma = 17$. Green halos mark the vertices in a common cluster with t.

[3] J. Alber and J. Fiala. Geometric separation and exact solutions for the parameterized independent set problem on disk graphs. *Journal of Algorithms*, 52(2):134–151, Aug. 2004.

[4] M. Ben Chen, S. J. Gortler, C. Gotsman, and C. Wormser. Distributed computation of virtual coordinates for greedy routing in sensor networks. *Discrete Applied Mathematics*, 159(7):544–560, Apr. 2011.

[5] P. Bose and P. Morin. Online routing in triangulations. *SIAM Journal on Computing*, 33(4):937–951, Aug. 2004.

[6] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent advances in graph partitioning. In L. Kliemann and P. Sanders, editors, *Algorithm Engineering*, volume 9220 of *Lecture Notes in Computer Science*, chapter 4, pages 117–158. Springer, Cham, 2016.

[7] I. Caragiannis and C. Kalaitzis. Space lower bounds for low-stretch greedy embeddings. *Theoretical Computer Science*, 610(Part B):149–157, Jan. 2016.

[8] P. Carmi, M. K. Chiu, M. J. Katz, M. Korman, Y. Okamoto, A. van Renssen, M. Roeloffzen, T. Shiitada, and S. Smorodinsky. Balanced line separators of unit disk graphs. *Computational Geometry*, 86, Jan. 2020.

[9] R. Dhandapani. Greedy drawings of triangulations. *Discrete & Computational Geometry*, 43(2):375–392, Mar. 2010.

[10] M. Dom. Compact routing. In D. Wagner and R. Wattenhofer, editors, *Algorithms for Sensor and Ad-hoc Networks*, volume 4621 of *Lecture Notes in Computer Science*, pages 187–202. Springer, Berlin, Heidelberg, 2007.

[11] R. Flury, S. V. Pemmaraju, and R. Wattenhofer. Greedy routing with bounded stretch. In *Proceedings of the 28th International Conference on Computer Communications*, INFOCOM 2009, pages 1737–1745, Rio de Janeiro, Apr. 2009. IEEE, New Jersey.

[12] J. Fox and J. Pach. Separator theorems and Turán-type results for planar intersection graphs. *Advances in Mathematics*, 219(3):1070–1080, Oct. 2008.

[13] G. N. Frederickson and R. Janardan. Efficient message routing in planar networks. *SIAM Journal on Computing*, 18(4):843–857, Aug. 1989.

[14] E. T. Gawlinski and H. E. Stanley. Continuum percolation in two dimensions: Monte Carlo tests of scaling and universality for non-interacting discs. *Journal of Physics A: Mathematical and General*, 14(8):L291–L299, Aug. 1981.

[15] A. George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973.

[16] A. Gupta, A. Kumar, and R. Rastogi. Traveling with a pez dispenser (or, routing issues in MPLS). *SIAM Journal on Computing*, 34(2):453–474, 2005.

[17] P. Gupta and P. R. Kumar. Critical power for asymptotic connectivity in wireless networks. In W. M. McEneaney, G. G. Yin, and Q. Zhang, editors, *Stochastic Analysis, Control, Optimization and Applications*, Systems & Control: Foundations & Applications, chapter 33, pages 547–566. Birkhäuser, Boston, 1999.

[18] H. Kaplan, W. Mulzer, L. Roditty, and P. Seiferth. Routing in unit disk graphs. *Algorithmica*, 80(3):830–848, Mar. 2018.

[19] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998. — The METIS source code is available at `http://glaros.dtc.umn.edu/gkhome/views/metis`.

[20] F. Kuhn, R. Wattenhofer, Y. Zhang, and A. Zollinger. Geometric ad-hoc routing: Of theory and practice. In *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 63–72, Boston, 2003. Association for Computing Machinery, New York.

[21] T. Leighton and A. Moitra. Some results on greedy embeddings in metric spaces. *Discrete & Computational Geometry*, 44(3):686–705, Oct. 2010.

[22] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[23] H.-I. Lu. Improved compact routing tables for planar networks via orderly spanning trees. *SIAM Journal on Discrete Mathematics*, 23(4):2079–2092, 2010.

[24] Y. Mao, F. Wang, L. Qiu, S. Lam, and J. Smith. S4: Small state and small stretch compact routing protocol for large static wireless networks. *IEEE/ACM Transactions on Networking*, 18(3):761–774, June 2010.

[25] W. Mulzer and M. Willert. Compact routing in unit disk graphs. In Y. Cao, S.-W. Cheng, and M. Li, editors, *31st International Symposium on Algorithms and Computation (ISAAC 2020)*, volume 181 of *Leibniz International Proceedings in Informatics (LIPIcs)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, 2020.

[26] C. H. Papadimitriou and D. Ratajczak. On a conjecture related to geometric routing. *Theoretical Computer Science*, 344(1):3–14, Nov. 2005.

[27] M. Penrose. *Random Geometric Graphs*, volume 5 of *Oxford Studies in Probability*. Oxford University Press, Oxford, 2003.

[28] D. A. Spielman and S.-H. Teng. Spectral partitioning works: Planar graphs and finite element meshes. *Linear Algebra and its Applications*, 421(2–3):284–305, Mar. 2007.

[29] M. Thorup. Compact oracles for reachability and approximate distances in planar digraphs. *Journal of the ACM*, 51(6):993–1024, Nov. 2004.

[30] M. Thorup and U. Zwick. Compact routing schemes. In *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '01, pages 1–10. Association for Computing Machinery, New York, 2001.

[31] F. Xue and P. R. Kumar. The number of neighbors needed for connectivity of wireless networks. *Wireless Networks*, 10(2):169–181, Mar. 2004.

[32] C. Yan, Y. Xiang, and F. F. Dragan. Compact and low delay routing labeling scheme for Unit Disk Graphs. *Computational Geometry*, 45(7):305–325, Aug. 2012.