UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Reactive Construction of Planar Overlay Graphs on Unit Disk Graphs

Master Thesis

for the Degree of Master of Science (M.Sc.)
in Course of Studies Computer Science

submitted by
Freya Surberg

Primary Reviewer:	Prof. Dr. Hannes Frey Institute for computer science
Secondary Reviewer:	Florentin Neumann, M. Sc. Institute for computer science

Koblenz, February 26, 2015

Erklärung

Hiermit bestätige ich, dass die vorliegende Arbeit von mir selbständig verfasst wurde und ich keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet–Quellen – benutzt habe und die Arbeit von mir vorher nicht in einem anderen Prüfungsverfahren eingereicht wurde. Die eingereichte schriftliche Fassung entspricht der auf dem elektronischen Speichermedium (CD-Rom).

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Zusammenfassung

Geographisches Cluster-basiertes Routing ist ein aktueller Ansatz wenn es um das Entwicklen von effizienten Routingalgorithmen für drahtlose ad-hoc Netzwerke geht. Es gibt bereits eine Anzahl an Algorithmen, die Nachrichten nur auf Basis von Positionsinformationen durch ein drahtloses ad-hoc Netzwerk routen können. Darunter befinden sich sowohl Algorithmen, die auf das klassische Beaconing setzen, als auch Algorithmen, die beaconlos arbeiten (keine Informationen über die Umgebung werden benötigt, außer der eigenen Position und der Position des Ziels). Geographisches Routing mit Auslieferungsgarantie kann auch auf Overlay-Graphen durchgeführt werden. Bisher werden die dafür benötigten Overlay-Graphen nicht reaktiv konstruiert. In dieser Arbeit wird ein reaktiver Algorithmus, der Beaconless Cluster Based Planarization Algorithmus (BCBP), für die Konstruktion eines planaren Overlay-Graphen vorgestellt, der die benötigte Anzahl an Nachrichten für die Konstruktion eines planaren Overlay-Graphen, und demzufolge auch Cluster-basiertes geographishes Routing, deutlich reduziert. Basierend auf einem Algorithmus für Cluster-basierte Planarisierung, konstruiert er beaconlos einen planaren Overlay-Graphen in einem unit disk graph (UDG). Ein UDG is ein Modell für ein drahtloses Netzwerk, bei dem alle Teilnehmer den gleichen Senderadius haben. Die Evaluierung des Algorithmus zeigt, dass er wesentlich effizienter ist, als die Baecon-basierte Variante. Ein weiteres Ergebnis dieser Arbeit ist ein weiterer beaconloser-Algorithmus (Beaconless LLRAP (BLLRAP)), für den zwar die Planarität aber nicht die Konnektivität nachgewiesen werden konnte.

Abstract

Geographic cluster based routing in ad-hoc wireless sensor networks is a current field of research. Various algorithms to route in wireless ad-hoc networks based on position information already exist. Among them algorithms that use the traditional beaconing approach as well as algorithms that work beaconless (no information about the environment is required besides the own position and the destination). Geographic cluster based routing with guaranteed message delivery can be carried out on overlay graphs as well. Until now the required planar overlay graphs are not being constructed reactively. This thesis proposes a reactive algorithm, the Beaconless Cluster Based Planarization (BCBP) algorithm, which constructs a planar overlay graph and noticeably reduces the number of messages required for that. Based on an algorithm for cluster based planarization it beaconlessly constructs a planar overlay graph in an unit disk graph (UDG). An UDG is a model for a wireless network in which every par-

ticipant has the same sending radius. Evaluation of the algorithm shows it to be more efficient than the non beaconless variant. Another result of this thesis is the Beaconless LLRAP (BLLRAP) algorithm, for which planarity but not continued connectivity could be proven.

Contents

List of Figures

Chapter 1

Introduction

Wireless ad-hoc sensor networks usually consist of a number of (mobile) nodes that communicate wireless with each other. A node is a mini computer that can be equipped with any number of sensors, for example GPS or temperature. Since nodes can be mobile or obstacles could have been put between them, connections between nodes can shift over time. While a connection might have existed at some point in time it cannot be assumed that it exists always. Thus, connections between nodes need to be determined every time a message is to be sent (ad-hoc). Two models for wireless ad-hoc networks are unit disk graphs (UDGs) and quasi unit disk graphs (QUDGs). Whereas in an UDG every node has the same consistent sending and receiving radius, this is not the case in a QUDG. The QUDG model allows for two different radii. One that is the same consistent radius for all nodes in the graph, where messages can be sent and received in any case, and one that can be different for every node and shift with time. In that radius messages only might be received.
Applications of wireless ad-hoc sensor networks can be building automation or the detection of problematic changes in an environment, like traffic jams or forest fires. To that purpose data from various nodes is collected in a commonly named data-sink, which computes whether an alarm has to be sent out (or another kind of action to be taken). Since the battery power of sensor nodes is rather limited, and therefore their transmitting radius, often a message cannot be send directly from source to destination, but has to be sent over several intermediate nodes. This is called multi-hop communication. To which node a message should be forwarded next is determined by a routing protocol. Classical routing tables are not efficient since they would need to be changed every time a connection is omitted or a new connection arises. Two general approaches for wireless ad-hoc networks are Greedy routing and Geographic Greedy routing. With Greedy routing the message is forwarded to the node which, after a specified metric, has the greatest gain. Applied metrics are for example shortest path

to destination or lowest cost (battery power). Geographic Greedy routing assumes that every node knows its own geographic location (for example via GPS). A message is forwarded to the node which is nearest to the destination (no routing tables necessary). Basic Geographic Greedy routing has the problem that message delivery cannot be guaranteed due to the possible existence of concave nodes. Concave nodes are nodes that have no outgoing connection to another node nearer to the destination. Routing backwards is usually not allowed, to avoid routing loops. One solution to that problem is planar graph routing (for example the FACE protocol [2]). If a message is sent to a concave node through Geographic Greedy routing, planar graph routing algorithms can take over as recovery strategy and thus ensure guaranteed message delivery. One type of planar graph routing algorithms are geographical cluster based routing algorithms. A key aspect of geographical cluster based routing algorithms is the grouping of nodes into clusters. Nodes are clustered according to their geographic position and their sending radius. One node per cluster is determined as clusterhead. This is often the node nearest to the middle of the cluster. Outwardly the clusterhead of a cluster often lies directly in the middle (its a virtual node). Edges between clusters, i.e their clusterheads, where an edge exists if two nodes in different clusters form an edge, belong to an overlay graph, which is called the aggregated graph [7]. An overlay graph is a subgraph of the physical graph. It contains less edges. Geographical cluster based routing algorithms route messages over planar, i.e. the graph has no intersections, cluster based overlay graphs. Routing over cluster based overlay graphs has the advantage, that edges between clusters tend to be more lasting than edges between (mobile) nodes [8]. As long as at least one node has an edge to a node in another cluster, the cluster edge exists as well.
Routing algorithms for wireless ad-hoc networks are required to be energy efficient, since battery power of wireless nodes is severely limited. The primary drain on that battery power is the sending of messages. In contrast, the cost of local computations is minimal.[18] Minimizing the number of messages that need to be sent to determine all current outgoing edges of a node is therefore a key factor when trying to increase efficiency of routing algorithms. Classic routing algorithms for wireless ad-hoc sensor networks rely on so called beacon messages. A beacon message is periodically send out by each node in the network to announce its continued existence. Connections between nodes can be determined using those beacon messages. A node has a connection to every other node it receives beacon messages from. Disadvantages of beacon messages are that they cause considerable traffic in the network, and therefore a higher chance of interference, and that substantial energy is needed to send them. Consequently, the research of beaconless routing algorithms has intensified. Beaconless algorithms do without beacon messages. Nodes only send messages if they have to, i.e. they either want to have information from another node or they answer a

specific request from one. This reduces the number of sent messages and therefore interference as well as power usage considerably. The majority of beaconless algorithms work local. In the context of this thesis the term "reactive" will mean beaconless as well as local.
Although geographical cluster based routing algorithms offer guaranteed delivery and are applicable in various scenarios, til now no beaconless algorithm to construct a planar overlay graph existed. This thesis closes that gap. With the development of the Beaconless Cluster Based Planarization (BCBP) algorithm, it is now possible to reactively construct a planar overlay graph on UDGs. The BCBP algorithm is based upon an algorithm that planarizes UDGs using Gabriel Graph construction (more information can be found in Section 3.9). Besides proofing the correctness of the BCBP algorithm, giving pseudo code descriptions and ideas for improvements, this thesis evaluates the BCBP algorithm with a simulation. As expected, the algorithm reduces the number of messages needed to construct a planar overlay graph considerably (about 70%). The best performance is reached in high node densities. Another result of this thesis is a beaconless variant of the localized link removal and addition based planarization (LLRAP) algorithm (Beaconless LLRAP (BLLRAP)) (see Section 3.10 for more details). It could not be proven that the BLLRAP algorithm constructs connected overlay graphs. Planarity, though, is another matter, and could be proven (i.e. the property that leads to planarity could be proven).
This thesis is structured in the following way: the first two chapters deal with work related to this thesis and basics the following chapters rely on. Everything that is not directly used to produce the results of this thesis or needed to understand those results is contained in the related work chapter (most aspects described in the basics chapter can be considered related work as well). Following that, the developed algorithms are described in detail and seeral proofs are given. Chapter five presents the results of the simulation of the BCBP algorithm, followed up by a discussion of aforementioned results and further work proposals in chapter six.

Chapter 2

Related Work

This chapter contains an overview of work related to this thesis. It is divided into two sections, which deal with different aspects. The first section concentrates on works on overlay graphs, whereas the second section deals with reactive algorithms for geographic routing.

2.1 Overlay Graphs

An overlay graph is usually a subgraph of the original graph. While an overlay graph does not have to be planar per definition, planarity is often required. Thus, many algorithms focus on the extraction of planar overlay graphs. One method to construct a planar overlay graph is the extraction of a Gabriel Graph [11] from the original graph. Only edges that fulfill the Gabriel Graph condition are accepted into the Gabriel Graph (more on Gabriel Graphs in Section 3.7). In [14] a Gabriel Graph is used as a basis for planar graph routing. In [9] a planar subgraph is constructed by nodes being organized into geographic clusters which form the vertices of a Gabriel Graph. While the algorithm proposed in [9] may produce disconnected overlay graphs, the algorithm proposed by Frey et al. in [6] does not (for more details see Section 3.9).
A special kind of subgraph is a spanner (see Section 3.5.3). A spanner is a subgraph of an original graph that spans the whole original graph, i.e. it does not divide the graph into parts. Catusse presents in [3] an algorithm to construct a planar hop spanner for UDGs with a constant stretch factor. The stretch factor is the "worst factor by which distances increase" [18] in the overlay graph in comparison to the original graph. Constant stretch factors are preferable to not constant stretch factors, since assumptions can be made that way.
Several algorithms were devised to extract a planar subgraph from an QUDG. In [5] a "nearly planar backbone with a constant stretch factor" is the result of the proposed algorithm. A local algorithm to extract a sparse constant

spanner was defined by Lillis et al. in [15]. In contrast to "normal" spanners, sparse spanners have less nodes and edges and are therefore easier manageable (concerning routing decisions). Barriere et. al. [1] construct planar overlay graphs on QUDGs using Gabriel Graph construction. Since just using Gabriel Graph construction on an QUDG can lead to a disconnected overlay graph, the inclusion of virtual edges was proposed.

2.2 Reactive Algorithms

One algorithm to route messages based on position information beaconless through a wireless ad-hoc sensor network is the Implicit Geographic Forwarding (IGF) algorithm [20]. It works state-free, i.e. no information about the environment besides the own position and the destination of the message is required. Another algorithm is the contention-based forwarding scheme (CBF) [10]. It selects the next hop on route to the destination with the help of biased timers. Selecting one node as next hop (timer fired) stops the selecting of any other node (timers are stopped). The Beaconless Routing Algorithm (BLR) [13] works beaconless as well. To choose the next hop the message is broadcasted to all potential forwarding nodes in a specified area. Each node starts a timer dependent on their distance to the destination (Greedy routing). The node nearest to the destination forwards the message, thereby stopping all other timers. Another beaconless routing algorithm that is based on position information is the Guaranteed Delivery Beaconless Forwarding (GDBF) algorithm [4]. This algorithm combines geographic greedy routing with planar graph routing. Depending on the situation the message is either forwarded to the node nearest to the destination (greedy mode) or a Gabriel Graph neighbor of the forwarding node (recovery mode). In both cases candidate nodes are determined through timers.

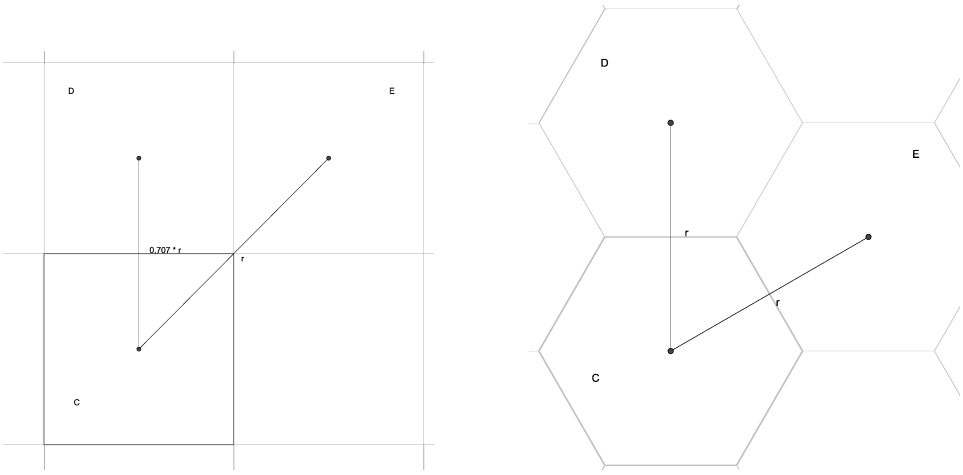
Chapter 3

Basics

This chapter contains basic information about concepts and algorithms on which the following chapters of this thesis are based upon.

3.1 Clustering

Algorithms for routing messages in a wireless network sometimes require clustering of nodes. There are different ways to cluster nodes. One way to cluster nodes is based upon the assumption that every node in a cluster has a connection to all other nodes in the cluster and preferably only a few to nodes in other clusters. Geographical clusters use, beside the sending radius of a node, the node's geographic location to assign nodes to clusters. In [7] Frey points out the advantages of partitioning the plane in regular polygons. Regular hexagons, squares and triangles are the only polygons that can be used to divide the hole plane gap-less into clusters.
Regular hexagons have the advantage that the distance between two adjacent hexagons (from one clusterhead to another) is always the same, which makes certain properties easier to proof. Figure 3.1 shows a comparison between a hexagon and a square grid. Whereas all hexagons have the same distance to each other, there are two different distances between neighboring clusters in the square grid. Each cluster in a hexagon grid can be unambiguously identified by its center. Starting from the cluster at the origin of the coordinates system (0/0), every cluster can be addressed by two vectors (where the origin is assumed to be the left upper corner). Every node can locally determine its own cluster with its own location information and the origin of the coordinate system. With a total ordering on the plane the assignment of clusters to nodes is unique.



(a) Square grid (b) Hexagon grid

Figure 3.1: Distances between two neighboring clusters for hexagons and squares

3.2 Geographical based routing algorithms

Geographical based routing algorithms are based upon the assumption, that each node knows its own location. This information is used to find a path through a graph. The advantage of such an algorithm is the fact that no routing tables need to be maintained. Most geographical based routing algorithms use a form of Greedy routing, where the node is forwarded in the direction that brings the message closest to the destination. To avoid routing loops, backwards routing is usually not allowed. This can however lead to a message being "stuck" with a concave node, so that message delivery cannot be guaranteed. In those cases planar graph routing algorithms on geographical clusters can be used to guarantee message delivery. One planar graph routing algorithm is FACE [2]. It traverses the faces of a planar subgraph in the general direction of the destination (clockwise or counterclockwise) until the original Greedy routing algorithm can take over again.

3.3 UDG/QUDG

An UDG is a model for a wireless sensor network. One of its key properties is the fact, that all sensor nodes are assumed to have the same sending and receiving radius. This allows for easier theoretical reasoning with UDGs. A more realistic model is the QUDG which is first mentioned in [1]. In contrast to UDGs, QUDGs consider not one but two radii. One in which a node can send and receive messages in any case (r_{min}) and another one where

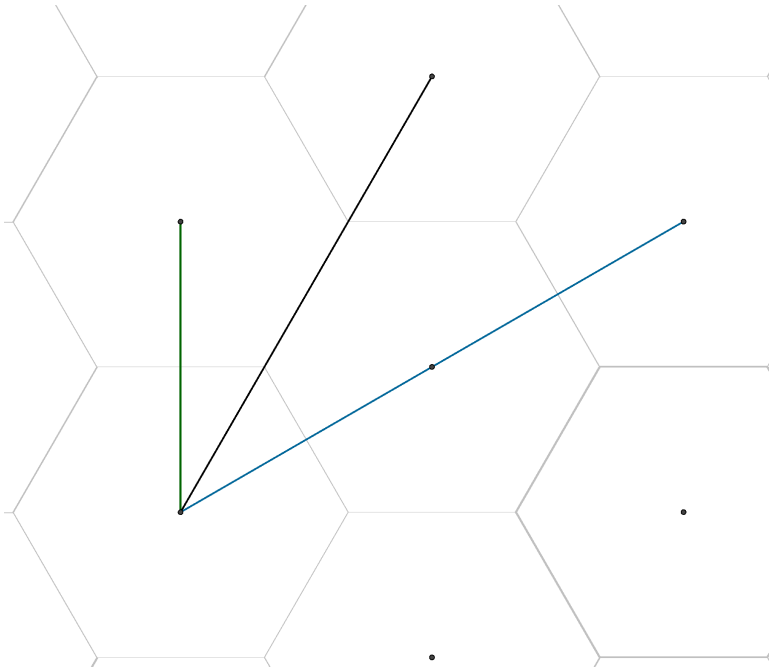
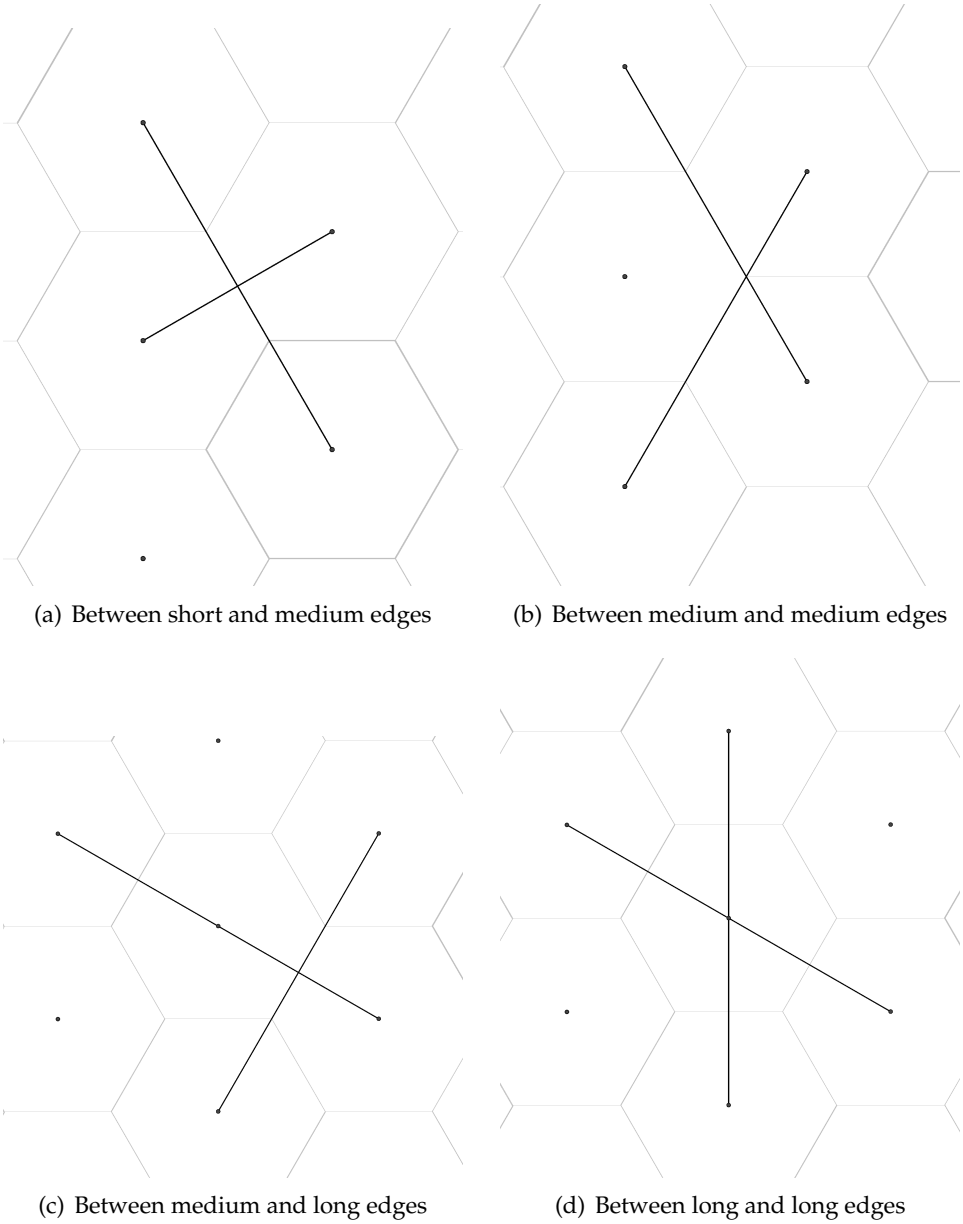


Figure 3.2: Types of edges in an UDG overlay graph

it may be able to send and receive messages(r_{max}). Thus, it is possible to model more natural circumstances. It was shown by Barriére et al that r_{min} and r_{max} should have a maximum ratio of $\frac{r_{max}}{r_{min}} \leq \sqrt{2}$ so that for all intersections in a QUDG at least one node of the edges forming an intersection can reach a node from the other edge and therefore detect the intersection with 2-hop neighborhood information.

3.4 Overlay graphs in UDGs and QUDGs

When constructing an overlay graph of an UDG based on a hexagon grid, the size of the regular hexagons are usually chosen such, that all nodes in a cluster can reach each other. This means the diameter of a regular hexagon matches the size of the unit disk radius. That fact limits the number of possible reachable clusters and the number of edge types that can occur between clusters. In [7] Frey describes those edge types. It is differentiated between short (green), medium (black) and long (blue) edges, which can be seen in Figure 3.2. Those edges can intersect with each other in a limited number of ways, shown in Figure 3.3. To preserve the property that every node in a cluster can reach all other nodes in the same cluster, with QUDGs the diameter of a regular hexagon matches the size of r_{min}, i.e. it is assumed $r = r_{min}$. It follows that an QUDG cluster could potentially reach more clusters than an UDG cluster and that another edge type (red),



(a) Between short and medium edges

(b) Between medium and medium edges

(c) Between medium and long edges

(d) Between long and long edges

Figure 3.3: Intersections between edges in an UDG overlay graph

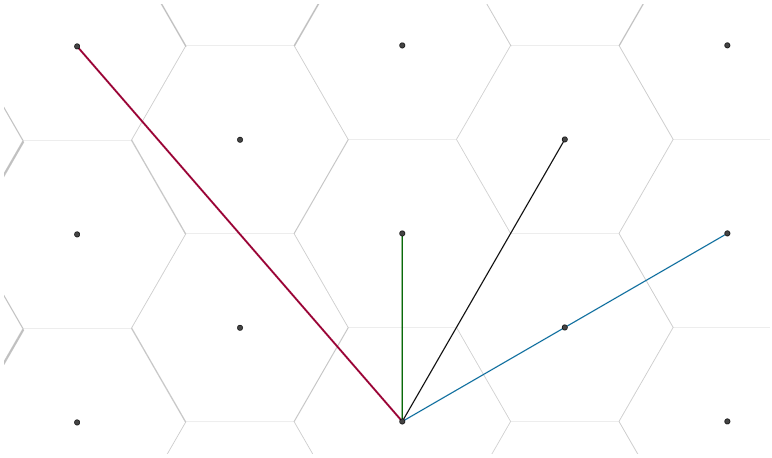


Figure 3.4: Types of edges in a QUDG overlay graph

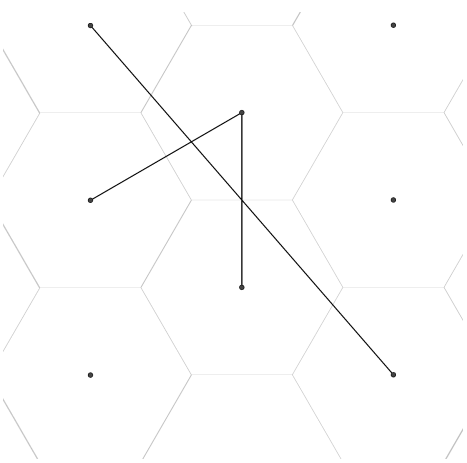
with more possibilities to intersect, can occur. This can be seen in Figures 3.4 and 3.5 In both graph models, edges can intersect in an irregular way. They are formed by a long edge and any other kind of edge that has an endpoint in the cluster the long edge passes through. Those intersections are called irregular intersections and can be seen in Figure 3.6. As can be seen, irregular intersections can only occur with long edges, since those are the only edges passing through cluster middles.

3.5 Graph Properties

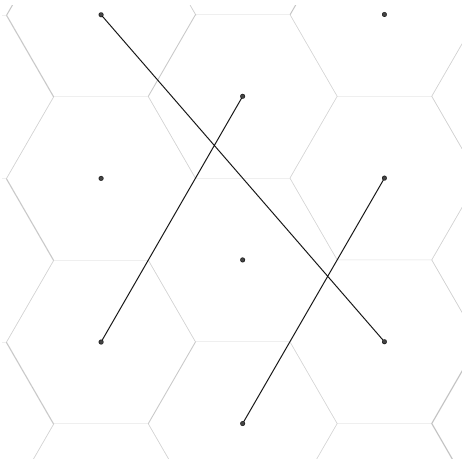
This section describes some graph properties that are relevant for this thesis.

3.5.1 Connectivity

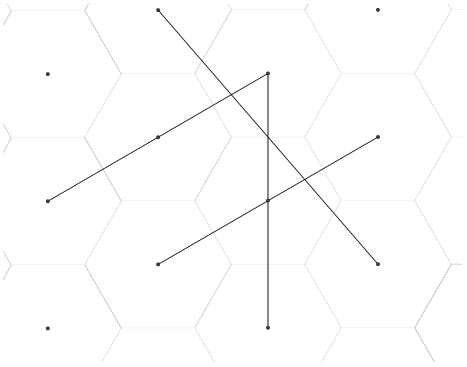
Connectivity is one of the key properties of graphs. A graph consists of nodes and edges between those nodes. If it is possible to find a way from one node to every other node in the graph using only existing edges, the graph is called connected. Otherwise the graph consists of several subgraphs. For some routing and recovery algorithms it is necessary, to construct an overlay graph of an original graph. If the original graph was connected it is important that the constructed overlay graph remains connected as well. Otherwise the results of routing in the original graph and routing in the overlay graph would differ concerning the success of delivering a message.



(a) Between short and overlong edges



(b) Between medium and overlong edges



(c) Between long and overlong edges

Figure 3.5: Intersections between edges in an QUDG overlay graph

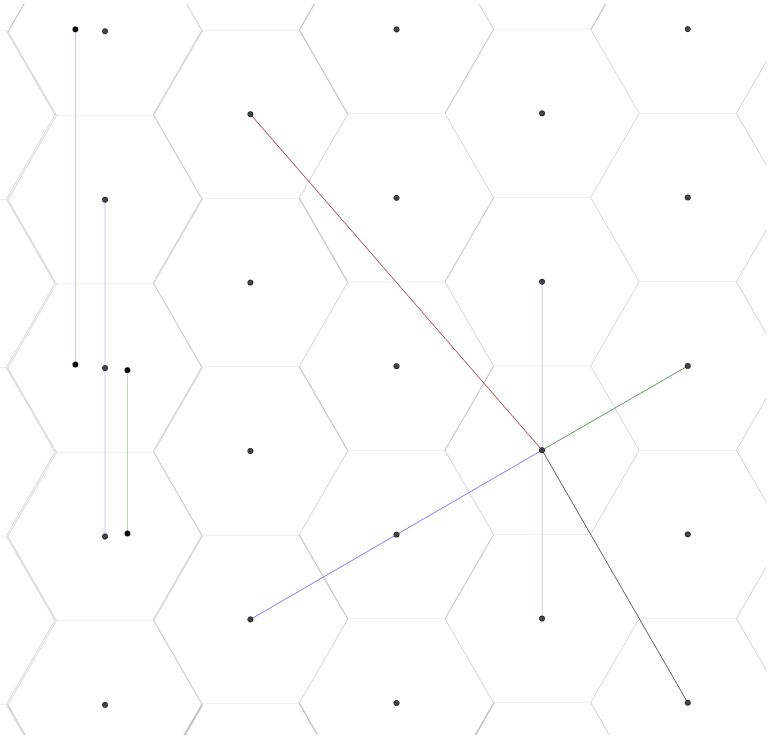


Figure 3.6: Types of irregular intersections in UDGs and QUDGs

3.5.2 Planarity

A graph is called planar if there are no intersections between straight line edges of embedded nodes. This property is important for algorithms that are used to recover from routing failures which may occur with algorithms like Greedy forwarding routing. Recovery algorithms based on geographical information usually explore the faces of an overlay graph to route a message. This can only be done if no intersections exist in the overlay graph, since routing loops could occur otherwise.

3.5.3 Spanners

A spanner is a subgraph of an original graph that spans the whole original graph, i.e. it does not divide the graph into parts. One possibility to measure the quality of a subgraph, is the spanning ratio. The spanning ratio is the "worst factor by which distances increase" [18] in the subgraph in comparison to the original graph. There are different kinds of spanning ratios. The two that are used the most are the euclidean spanning ratio and the hop spanning ratio. With the euclidean spanning ratio, the length of the shortest path is measured as the accumulated euclidean distance between the nodes on that path. The hop spanning ratio considers the length of a

path as the number of hops that need to be taken from start to end. In general the spanning ratio should be as low as possible. A general definition of the spanning ratio with graph $G = (V, E)$ and a spanning subgraph $G' = (V, E')$ with $E' \subseteq E$, is given in [18] as follows:

$$Stretch(G') = \max_{v,w \in V} \{ \frac{dist_{G'}(u, w)}{dist_G(u, w)} \}$$

3.5.4 k-neighborhood

The k-neighborhood of a node v in a graph is defined as all nodes reachable by v with at most k-steps. For example the 2-hop neighborhood of a node v consists of all nodes that can be reached in one or two hops.

3.6 Beaconless Algorithms

Sensor nodes only have a limited amount of battery power, so it is important that that power is conserved as much as possibly. Since sending messages is the most power consuming action of a sensor node, reducing the number of messages to be send is a good approach to save energy. In classical algorithms each sensor node periodically sends out a so-called beacon message. This message contains information about the nodes position and confirms at the same time that it still exists. Beaconless algorithms do without such beacon messages. Nodes only send messages if they have to, i.e. they either want to have information from another node or they answer a specific request from one. Getting information from a whole x-hop neighborhood should only be a worst case scenario with those kinds of algorithms. A key element of beaconless algorithms are timers. They can be used to structure the course of events i.e. a node is waiting a specified amount of time before continuing with the next step of the computation, thus avoiding that requests have to be answered. Another application of timers is delaying answering to requests. The amount of time an answer is delayed may depend on different criteria, for example the distance to the requesting node or the distance to the destination a message needs to be forwarded to. If a node overhears a response from another node to a request it started a timer for, it decides whether it can stop its own timer or needs to do anything else. Those kinds of timers are used to reduce the number of answering nodes and at the same time find a node that satisfies a certain condition, for example being nearest to the requesting node. Due to their nature beaconless algorithms are almost always local. In the context of this master thesis a reactive algorithm is considered to be beaconless as well as local.

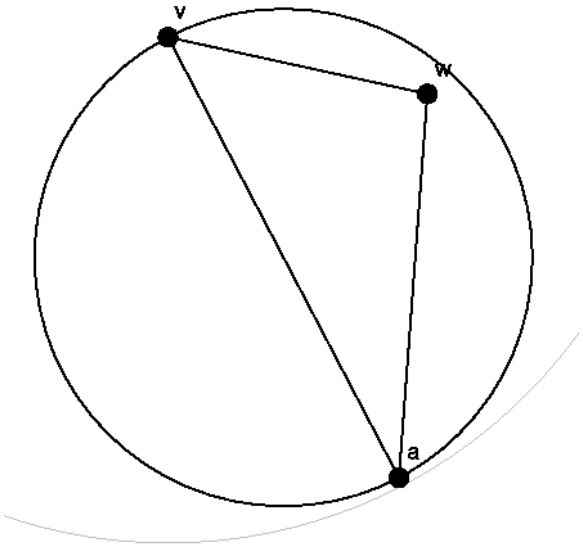


Figure 3.7: Proximity region of a Gabriel Graph

3.7 Gabriel Graph

Given a set of sensor nodes S in the plane and the corresponding graph G consisting of those nodes and the edges between them, a Gabriel Graph $GG(S)$ is a subset of edges in G. An edge of G, and the corresponding nodes, are in $GG(S)$ if and only if no other node lies in the circle $U(a, b)$ (including the border) which is also called the proximity region. [11] The diameter of the circle is $\overline{ab}$. Figure 3.7 shows a Gabriel Graph proximity region between the nodes v and a. Since node w lies in that region the edge va is not a Gabriel Graph edge. The edge va is a Gabriel Graph edge. As shown in [7] a Gabriel Graph is always planar and, if the original graph was connected, connected as well.

3.8 Beaconless Forwarder Planarization

The Beaconless forwarder planarization (BFP) algorithm [19] is a beaconless and local algorithm with which subgraphs based on proximity regions can be constructed from a source graph. One possible proximity region to be used is the Gabriel Graph condition. The algorithm consists of two phases. In the first phase, which is called the selection phase, the initiating node collects possible Gabriel Graph neighbors. In the second phase, which is called the protest phase, other nodes can protest against those possible neighbors. At the start the initiating node v sends out an RTS including its

own position. Every node that receives that message and therefore could be a Gabriel Graph neighbor of v, starts a timer depending on its distance to v. The shorter the distance the smaller the timer. When the timer expires the node sends an CTS to v. All other nodes overhearing that CTS stop their own timers, if the Gabriel Graph condition is not fulfilled for them. Those nodes are called hidden nodes. It can happen in this phase, that a node believes itself to be still a possible Gabriel Graph neighbor because the only node(s) lying in the proximity region already stopped its/their timer(s) due to another node answering first. In this case the hidden nodes add the false candidate to their protest list. After the timer of v for the execution of the first phase expires, it sends another message indicating that its ready to receive protests against Gabriel Graph candidate nodes. All hidden nodes, whose protest list is not empty, thereupon start a timer depending on their distance to v again. When the timer expires they send their protest list to v. Every node that receives such a protest list and whose timer did not yet expire, cross checks its own list against it and removes possible double entries from its own list. After the second phase is concluded v has a complete view of all its outgoing edges in the Gabriel Graph. In Figure 3.8 an example constellation of nodes is shown where a protest message is necessary. Node v starts the algorithm. Since u is the nearest node it answers first. Node w notices this and since u would lie in the proximity region between w and v stops its own timer. It is now a hidden node. The next node to answer is a, although w lies in the proximity region between v and a. w consequently adds a to its protest list and sends a protest response to v. The only Gabriel Graph edge of v in this scenario is u.

3.9 Planarization based upon Gabriel Graphs

In [6] Frey describes an algorithm that can be used to construct a planar and connected overlay graph. Since that algorithm already works locally, it only has to be modified to work beaconless as well to fulfill the requirement of a reactive algorithm.
The algorithm is a variant of the geographic cluster routing (GCR) algorithm, first mentioned in [9], and was developed to eliminate the problem of possible disconnectivity when applying GCR. GCR is based on the assumption that each node belongs to an unique geographical cluster, which can be determined by a node using its location information. The resulting overlay graph, in which an edge in the overlay graph exists if and only if an edge between two nodes in different clusters exists, is called the aggregated graph $H(G)$. This concept is called node aggregation and discussed in detail in [7]. To construct a planar overlay graph GCR uses a cluster based form of Gabriel Graph construction. If at least one of the clusters C or D is connected to another cluster in the circle $U(C, D)$ the edge CD is removed.

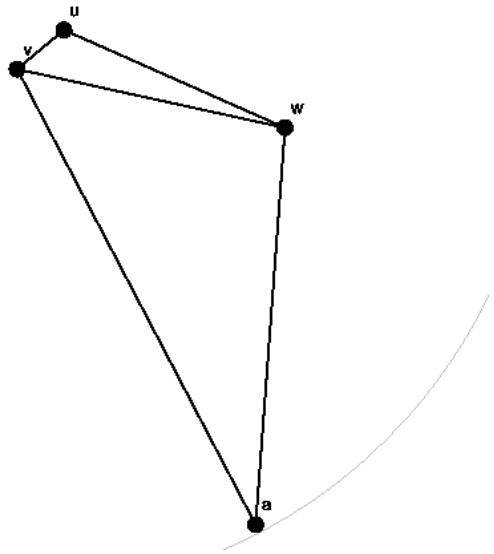


Figure 3.8: BFP algorithm - protest necessary

This method of constructing an overlay graph can lead to disconnectivity. Frey's variant of GCR does it the other way around. Instead of constructing an overlay graph by node aggregation and then eliminating all edges that do not fulfill the modified Gabriel Graph condition, the Gabriel Graph is constructed first (with the normal Gabriel Graph condition) and used as a basis for the overlay graph. The resulting overlay graph is called the *aggregated Gabriel Graph* $H(G(V))$. Frey shows in [6] that the aggregated Gabriel Graph of an UDG retains connectivity and is planar, except for irregular intersections (see 3.4). Those are taken care of whilst every long edge forming an irregular intersection is replaced by two short edges. The short edges replacing the long edge are called implicit edges, whereas all other short and medium edges as well as long edges not forming an irregular intersection are called explicit edges. Short edges may be implicit as well as explicit. If they are only implicit they are called pure implicit edges. Long edges that form an irregular intersection and are replaced by short edges are called removed edges.
The overlay graph is constructed in two steps. First every node constructs an unique view on its outgoing edges to other clusters. In a second step the outgoing edges of all nodes in a cluster are brought together to form an overall view. The algorithm for determining edges of a node v to nodes in other clusters distinguishes four sets of edges. $E(v)$ contains all explicit edges, $I_s(v)$ and $I_r(v)$ contain the implicit edges in the same and in the reverse direction respectively, whereas $R(v)$ contains all removed long edges. In Figure 3.9 the different types of edges are shown. The edge uw results

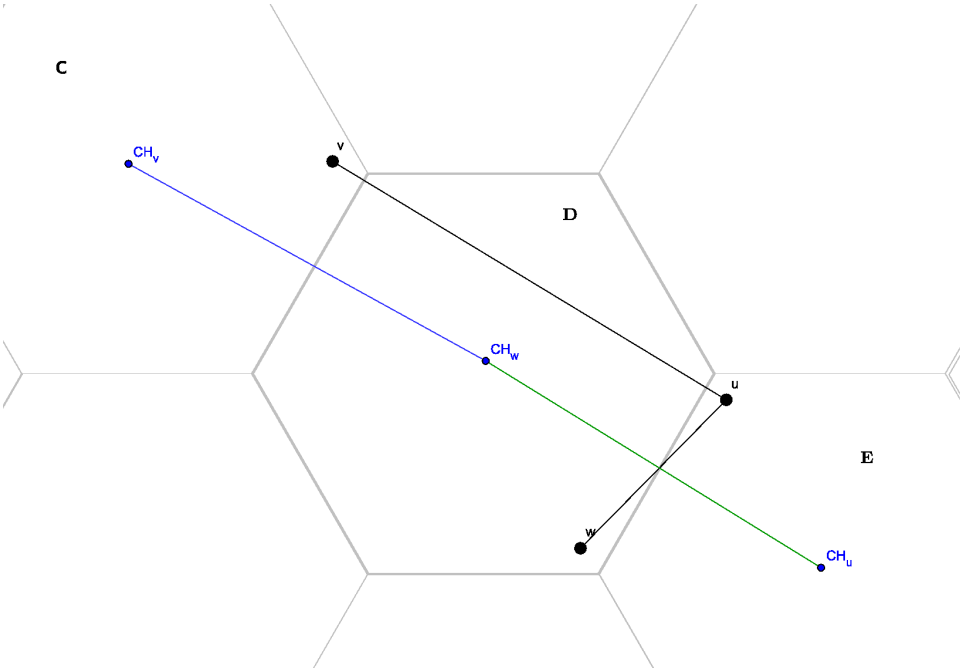


Figure 3.9: Types of edges

in the green overlay edge DE, which is explicit. The blue overlay edge CD between the clusters of v and w is an implicit edge. From $v's$ point of view it is an implicit edge in the same direction, i.e. to send a message to w it first has to route it in the same direction over the long edge to u which then routes it to w. From $w's$ point of view it is an implicit edge in reverse direction. It has to route a message intended for v in the reverse direction over the short edge to u first. In a complete overlay graph implicit edges in the same and reverse direction always come in pairs. The sets are computed in the following way:

- For all neighbors u of v in the Gabriel Graph
  - if a cluster D exists that can be reached by v or u and lies in the middle of C (cluster of v) and E (cluster of u)
  - then add the edge CE to $R(v)$
    - if D can't be reached from C
    - then add CD to $I_s(v)$
  - else add CE to $E(v)$
- For all neighbors u of v in the UDG

- for all removed long edges DE in $R(u)$ of u where D is the cluster of any node
  - if C lies between D and E
  - then
    - if D can't be reached from C by any node
    - then add CD to $I_r(v)$

For all its neighbors $w \in V$ in $G(V)$, for which E denotes the cluster of w, it is checked whether a cluster D, that lies between C and E, exists and can be reached by either v or w. If this is the case an irregular intersection exists at this point and the long edge CE is not a candidate for an edge in the aggregated Gabriel Graph $H(G(V))$. An irregular intersection can only be formed with long edges. To determine whether the edge CD is a pure implicit edge, it is checked whether the cluster D is reachable by any node of cluster C. Should that be the case the edge CD is added to the set $I_s(v)$. All short and medium Gabriel Graph edges as well as long Gabriel Graph edges which do not form an irregular intersection are added as explicit edges to $E(v)$. After this step $E(v)$ contains all short and medium edges as well as long edges which do not form an irregular intersection and $I_s(v)$ contains all pure implicit edges with one hop in the same direction. In the next step all pure implicit edges of a node v with one hop in the reverse direction are computed. For every neighbor $w \in V$ in $U(V)$, for which E denotes the cluster of w, it is checked whether w has a removed long edge DE, as computed in the step before, that crosses over C. If this is the case and D can not be reached by any node of C then the edge CD is a pure implicit edge with one hop in the reverse direction of v and added to $I_r(v)$. After this step $I_r(v)$ contains all pure implicit edges with one hop in the reverse direction of v. After every node in cluster C calculated the sets $E(v)$, $I_s(v)$ and $I_r(v)$ they are shared with all other nodes in the cluster and merged locally. The result are the sets $E(C)$, $I_s(C)$ and $I_r(C)$ which provide an overall view on all outgoing edges of a cluster.

3.10 LLRAP

With the LLRAP algorithm [16] graphs more general than UDGs can be planarized. It can be applied to every graph which satisfies the two properties redundancy and coexistence. These properties are defined in [16] the following way:

Definition 1. *A graph satisfying redundancy property has, for any two intersecting edges, at least one node of the intersecting edges is directly connected to the*

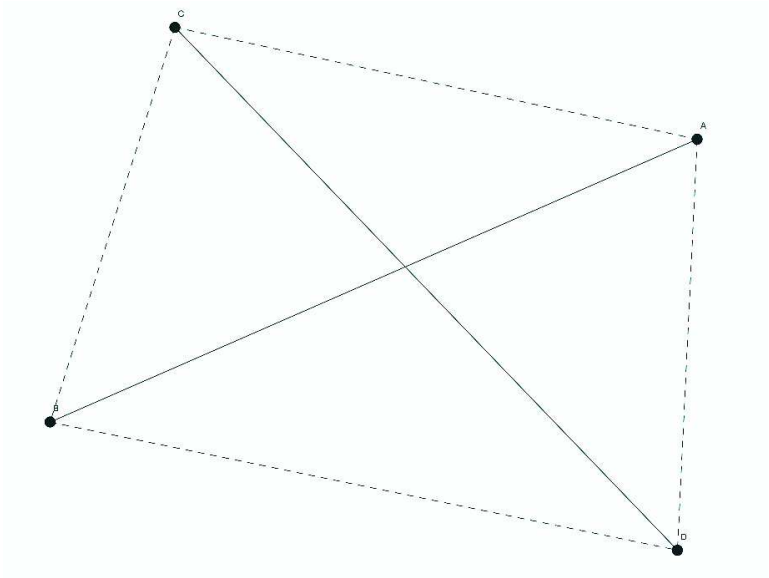


Figure 3.10: Redundancy property

remaining three nodes of the intersecting edges.

Definition 2. *A graph satisfying coexistence property has, for any three existing edges uv, vw and wu in the graph, if there is a node x lying inside the triangle Δuvw, then the edges ux, vx and wx also exist in the graph.*

Figure 3.10 shows the meaning of the redundancy property. Of the four nodes A, B, C and D forming the two intersecting edges AB and CD, at least one has to have connections to the remaining two nodes. Meaning the edges AC and AD, BC and BD, CB and CA, or DA and DB have to exist as well. The meaning of the coexistence property can be seen in Figure 3.11. If three nodes A, B, and C are all connected with each other the edges form a triangle. Any node lying inside that triangle, for example D, has to have connections to A, B and C as well. The algorithm consists of two phases. In the first phase some edges that form intersections are removed. For this purpose each node checks if it can detect an intersection in its one-hop neighborhood and if an alternate path exists to connect the endpoints of a removed edge. This always leads to all intersecting edges being removed, since redundancy property guarantees that at least one node of two intersecting edges can detect that intersection and a possible alternate path. After the first phase the resulting overlay graph may be disconnected although the underlying original graph was connected. To remedy that problem, in the second phase of the algorithm some of the removed edges may be added again. To decide which edges can be added without loosing planarity of the overlay graph, each node considers whose of its removed edges could be candidates for addition. If both nodes of a removed edge

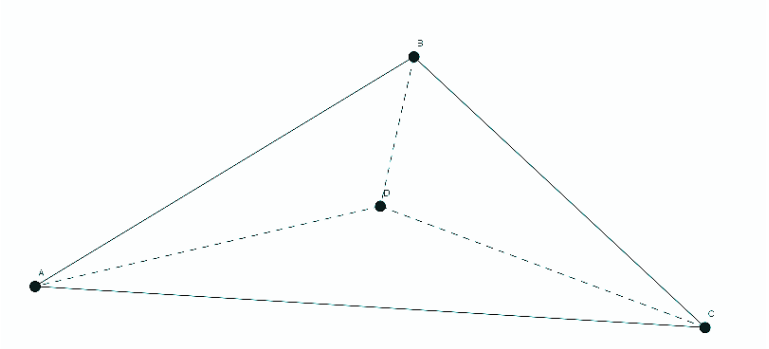


Figure 3.11: Coexistence property

consider the addition of the edge unproblematic (no intersection follows from adding the edge), it is added again.
The coexistence property guarantees that the overlay graph of an originally connected graph remains connected.

3.11 Beaconless Clustering Algorithm

This section contains a brief description of the Beaconless clustering algorithm (BCA) as well as an explanation of the alterations made to the algorithm.

3.11.1 Description

To route a message in an overlay graph a node needs to know all of its clusters outgoing edges. In [17] a beaconless clustering algorithm BCA is described which accomplishes that.
The algorithm consists of two phases. In the first phase the initial node v sends out a request, called first-request, to nodes in other clusters. Every node that receives this request and lies in another cluster as v, starts a timer depending on its distance to v. When the timer terminates, the node sends a response to v called first-response. All nodes that receive this response and lie in the same cluster as the answering node, stop their timer. This ensures that always the shortest connection to a cluster is chosen (unique). After a certain amount of time v starts the second phase of the algorithm, in which the other nodes in the cluster search for outgoing edges. For this purpose v first computes a list of all clusters to which no edge exists yet. As long as that list is not empty, v sends out a request, called second-request, to its internal cluster neighbors, to search for a connection to one of the clusters on the list. All internal neighbors then start a timer depending on their individual distance to the cluster. When the timer terminates, the

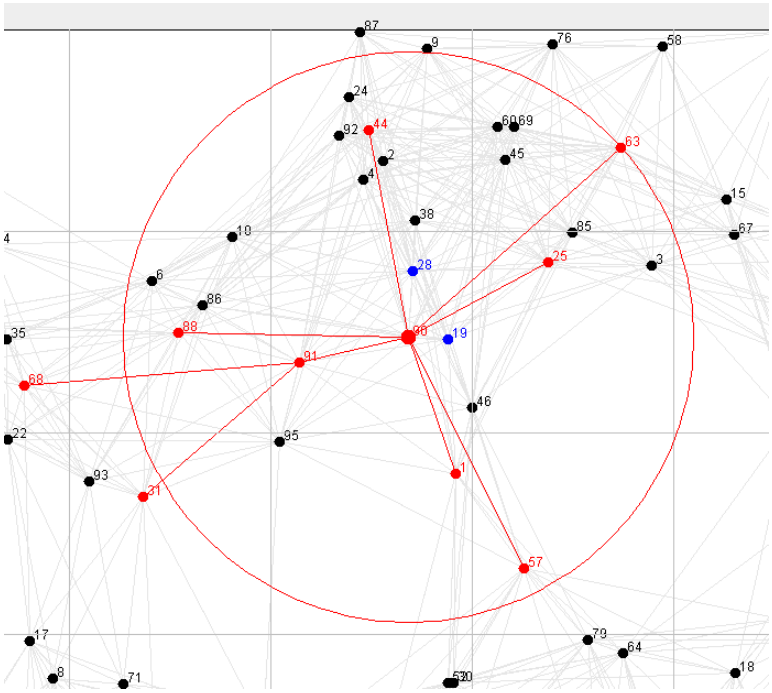


Figure 3.12: Result of the BCA

node sends a first-request of its own to that cluster and waits for a response. The other nodes in the cluster receiving the first request, pause their own timers. If no response is received the next node sends a first-request. If a response is received, the node sends a second-response message to v and the other nodes stop their timers. v removes the cluster from the list and sends the next request if open clusters remain. After the second phase of the algorithm v knows all outgoing edges of its cluster and whether or not it has to route over an intermediate node to reach certain clusters. The result of the algorithm can be seen in Figure 3.12. Red nodes indicate an existing outgoing edge whereas blue nodes are not needed to form any outgoing edge. After the algorithm is finished, the initiating node knows all outgoing edges to clusters that can be reached by at most two hops (either it can reach the cluster directly (one hop) or over an intermediate node in the same cluster (two hops)).

3.11.2 Alterations

3.11.2.1 Adaptation to hexagon grids

The BCA is based on square grids. For the adaptation to hexagon grids certain alterations had to be made:

- a cluster has now the form of a hexagon with the accompanying prop-

erties

- a cluster is now addressed by a unique cluster address instead of an ID
- all functions for geographical computations (for example whether a cluster is reachable by a certain node) were altered to take the new geometric conditions into account; to that purpose a library was introduced

Since the the grid now consists of hexagon clusters, the types of outgoing edges of a cluster have changed. Furthermore the euclidean distance between two adjacent clusters is now always the same. The address of a cluster is represented by three scalars. Each scalar stands for the number of times a specific direction vector has to be added to get from the origin of the plane (0/0) to the cluster. In null-positive-form, i.e. one scalar is zero while the other two are positive, each cluster address is unique. The library used for geometric computations is javaGeom-0.11.2. It includes models for lines as well as circles and offers methods to compute intersections between them. Whether a cluster is reachable from a certain node is determined by computing all intersections of the circle formed by the sending radius of the node and the edges of the destination hexagon. If at least one intersection exists, the node can reach the cluster. If no intersection exists, but the middle of the destination cluster is included in the sending radius of the node, the destination cluster lies completely inside the sending radius of the node and is therefore reachable as well.

3.11.2.2 Overlay graph

To see why the computation of all outgoing edges of a cluster does not suffice to construct an overlay graph that can be used for example for the FACE algorithm, an extra function that computes the outgoing edges for all clusters at once was implemented. The result can be seen in Figure 3.13. As evidenced the resulting overlay graph is not planar, which leads to the loss of guaranteed delivery message. The edges painted in orange indicate irregular intersections. Developing an algorithm that beaconless planarizes such an overlay is therefore necessary.

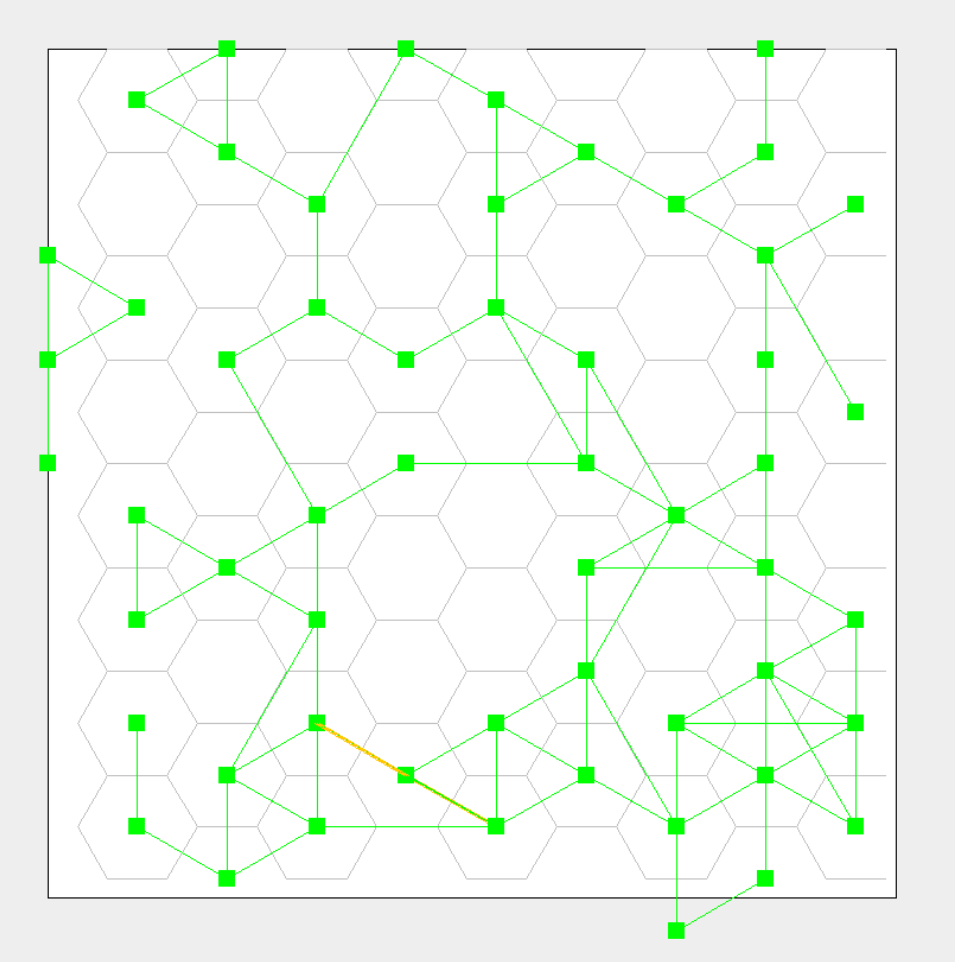


Figure 3.13: Overlay graph created when all clusters employ the algorithm used to determine all outgoing edges

Chapter 4

Algorithms

The goal of this thesis is the development of a beaconless algorithm that constructs a planar overlay graph in unit disk graphs in a reactive manner. Such an overlay graph can than be used as basis for geographical cluster based routing algorithms. In a first approach the LLRAP algorithm is examined for suitability, since its guarantees planarity and connectivity if a graph has two distinctive properties. Since one of those properties cannot be proven, another algorithm is developed. This algorithm is based upon the planar graph construction method described in Section 3.9 and will be called Beaconless Cluster Based Planarization algorithm (short: BCBP). The first section of this chapter explains which communication model is used for both algorithms. Following that a beaconless version of the LLRAP algorithm (see Section 3.10) called Beaconless LLRAP is presented, including all proofs that could or could not be made. The final section contains a description of the BCBP algorithm, along with a proof of correctness and suggestions for improvements.

4.1 Communication Model

One of the key factors of developing a distributed algorithm is the question which communication model to use. In [18] Peleg describes the two most often used models. These models lie on opposite ends of the synchronicity spectrum. Either a fully synchronous communication model is used or a totally asynchronous one. The fully synchronous model assumes that every node has a local clock which is synchronized with the clock of all other nodes, i.e. there is a global clock. The sending of a message takes less than one unit. Thus it can be deduced when a message should arrive at its destination and how long an answer might take. In contrast, the totally asynchronous communication model is event-based. It is not known how long a message might take from start to destination. Furthermore, messages may arrive in random order so that no overall state of the com-

putation can be deduced. None of the two models are very realistic, they are however suitable to generalize certain findings.
Since one key component of beaconless algorithms are timer, which are used for multiple purposes, like deciding whether to answer a query or how long to wait for an answer, the algorithms developed in this thesis use the fully synchronous communication model.

4.2 Beaconless LLRAP (BLLRAP)

The BLLRAP algorithm is based upon the LLRAP algorithm proposed by Frey and Mathews as described in Section 3.10. For the LLRAP algorithm to be applicable a graph has to have two properties: Redundancy property and Coexistence property. If it can be proven that the constructed overlay graph has those two properties, it will be inherently planar and connected. The first subsections of this section contain a description of BLLRAP as well as pseudo code descriptions of messages and timers used. The results of the attempt to prove Redundancy and Coexistence properties follows in the last subsection.

4.2.1 Description

The BLLRAP algorithm utilizes a number of messages and timers which are explained in detail in subsection 4.2.2 and subsection 4.2.3 respectively. The algorithm consists of the following five steps:

1. determine all outgoing edges of a cluster using the BCA described in Section 3.11
2. check for intersections and remove edges as necessary (removal step)
3. check whether some edges can be added again on this side (addition step)
4. check whether the additionable edges can be added from the other side as well
5. handle all irregular intersections that are left

In the first step all outgoing edges of a cluster are computed using the algorithm described in Section 3.11. In the following step it is checked whether any intersection exists. To that purpose the initial cluster sends an *CheckForIntersectionsRequest* to all its original neighbors. If those haven't computed their outgoing edges yet, that will be done first. Afterwards they check whether they can detect any intersections with the outgoing edges of the initial cluster. If that is the case an *IntersectionFoundResponse* is send

back to the initial cluster. The third step starts with a *CheckAdditionPossibleRequest* from the initial cluster. All removed edges are being investigated in regard to the question whether they could be added again. If a cluster that receives the request did not yet execute the removal step that will be done first. As soon as the removal step is finished the cluster checks for every edge whether an addition would lead to new (old) intersections. If an edge can't be added again an *AdditionNotPossibleResponse* is send to the initial cluster. In the fourth step the initial cluster checks for all removed edges which could potentially be added again, if they can be added from the other side as well. To that purpose a *CheckAdditionPossibleOtherSideRequest* is send to all endpoints of potentially additionable edges. There the addition step is executed and in case the edge can't be added again a corresponding *AdditionNotPossibleResponse* send back to the initial cluster. Finally the initial cluster computes all irregular intersections that are left.

4.2.2 Messages

This subsection contains a list of all messages used in the beaconless llrap algorithm. For each message a brief pseudo code description on how to process it is given.

FirstRequest

if *message for me* **then**
  start FirstResponseTimer;
  `// dependent on distance to cluster middle`
end
else if *message from this cluster* **then**
  if *ComputeOutgoingEdgesSecondStepTimer running* **then**
    suspend timer for $t_{max} + 2$;
  end
  else if *FirstRequestTimer running* **then**
    suspend timer for $t_{max} + 2 + timeleft$;
  end
end

FirstResponse

```
if message for me then
    if message needs to be forwarded then
        send SecondResponse (broadcast);
        remove answering cluster from list of open clusters;
    else
        add edge to list of outgoing edges;
    end
end
else if message from this cluster then
    if FirstResponseTimer running then
        stop timer;
    end
end
else if open clusters contains answering cluster then
    remove answering cluster from list of open clusters;
end
```

SecondRequest

```
if message from this cluster then
    start FirstRequestTimer;
    // dependent on distance to nearest cluster
       middle
end
```

SecondResponse

```
if message for me then
    add edge to list of outgoing edges;
end
else if message from this cluster then
    if list of open clusters contains edge then
        remove edge from list of open clusters;
    end
end
```

CheckForIntersectionsRequest

```
if message for me then
    if message from this cluster then
        forward request (broadcast);
    else
        if not this node computed outgoing edges then
            forward request (no broadcast);
        end
        else if not compute outgoing edges step started then
            send FirstRequest (broadcast);
            start ComputeOutgoingEdgesFirstStepTimer;
        end
        else
            compute intersections;
            send IntersectionsFoundResponse (no broadcast);
        end
    end
end
```

IntersectionsFoundResponse

```
if message needs to be forwarded then
    forward message;
else
    remove source from wait for answer list;
    remove intersections from list of outgoing edges;
    add intersections to list of removed edges;
    if not waiting on any more answers then
        if CheckAdditionPossibleRequest exists then
            compute intersections in case of addition;
            send AdditionNotPossibleResponse (no broadcast);
        else
            if any edges were removed then
                send CheckAdditionPossibleRequest (broadcast);
            else
                handleIrregularIntersections();
            end
        end
    end
end
```

CheckAdditionPossibleRequest

```
if message for me then
    if message from this cluster then
        forward request (broadcast);
    else
        if not this node computed outgoing edges then
            forward request (no broadcast);
        end
        else if not removal step started then
            add destinations to list of clusters to wait for;
            send CheckForIntersectionsRequest (broadcast);
        end
        else
            compute intersections in case of addition;
            send AdditionNotPossibleResponse (no broadcast);
        end
    end
end
```

AdditionNotPossibleResponse

```
if message needs to be forwarded then
    forward message;
else
    remove source from wait for answer list;
    add non additionable edges to list of non additionable edges;
    if not waiting on any more answers then
        if general computation then
            if any additionable edges then
                send CheckAdditionPossibleOtherSideRequest
                (broadcast);
            else
                handleIrregularIntersections();
            end
        end
        else if other side computation then
            if any edges additionable then
                add additionable edges to list of outgoing edges;
            else
                handleIrregularIntersections();
            end
        end
        else if general computation other side then
            send AdditionNotPossibleResponse (no broadcast);
        end
    end
end
```

CheckAdditionPossibleOtherSideRequest

```
if message for me then
    if message from this cluster then
        forward request (broadcast);
    else
        if not this computed outgoing edges then
            forward request (no broadcast);
        else
            add destinations to list of clusters to wait for;
            send CheckAdditionPossibleRequest (broadcast);
        end
    end
end
```

CheckForIrregularIntersectionRequest

```
if message from this cluster then
    forward request;
    // message is not broadcasted
else
    if not this node computed outgoing edges then
        forward request (no broadcast);
    else
        compute intersections;
        send IntersectionsFoundResponse (no broadcast);
    end
end
```

IrregularIntersectionFoundResponse

```
if message needs to be forwarded then
    forward message;
else
    remove long edge from list of open long edges;
    add edge to list of implicit edges in the same direction or remove
    from list of outgoing edges;
    if no open long edges exist then
        determine clusters to check for implicit edges in the reverse
        direction;
        if clusters to check then
            send ExistsLongEdgeOverMeRequest to each cluster on
            the list (no broadcast);
            // no broadcast so forwarding is possible
        else
            finished;
        end
    end
end
```

ExistsLongEdgeOverMeRequest

if *message from this cluster* **then**
  forward request;
  `// message was not broadcasted`
else
  if *not this node computed outgoing edges* **then**
    forward request (no broadcast);
  else
    send LongEdgeExistsResponse (no broadcast);
  end
end

LongEdgeExistsResponse

if *message needs to be forwarded* **then**
  forward message;
else
  remove cluster from list of clusters to check for implicit edges in the reverse direction;
  add edge to list of implicit edges in the reverse direction if necessary;
  if *no clusters to check* **then**
    finished;
  end
end

handleIrregularIntersections()

add all long edges to list of open long edges;
compute own irregular intersections;
if *any open long edges* **then**
  send CheckForIrregularIntersectionsRequest to each open long edge;
else
  determine clusters to check for implicit edges in the reverse direction;
  if *clusters to check* **then**
    send ExistsLongEdgeOverMeRequest to each cluster on the list (no broadcast);
    `// no broadcast so forwarding is possible`
  else
    finished;
  end
end

4.2.3 Timers

This subsection contains a list of all timers used in the BLLRAP algorithm. For each timer a brief pseudo code description of what happens when the timer fires is given as well as information about how their duration, and if necessary suspension, is determined. All timers are dependent on a variable called t_{max}, which is the maximum amount of time a timer can be initially set to, when contending for the right to answer a request.

- **ComputeOutgoingEdgesFirstStepTimer**
  This timer is a structural timer, i.e. it is used to structure the course of events. It is started when the first step of the BCA (finding outgoing edges directly) is started. Since the timer used to determine the node to answer the request (*FirstResponseTimer*) has a maximum duration of t_{max} and the time messages travel has to be considered as well (one unit for each request and response), the duration of the *ComputeOutgoingEdgesFirstStepTimer* is computed as follows:

  $$duration = t_{max} + 2$$

ComputeOutgoingEdgesFirstStepTimer
send SecondRequest; start computeOutgoingEdgesSeconsStepTimer;

- **ComputeOutgoingEdgesSecondStepTimer**
  This timer is a structural timer, i.e. it is used to structure the course of events. It is started when the second step of the BCA (finding outgoing edges over intermediate nodes) is started. Since the timer used to determine the node to forward the request (*FirstRequestTimer*) has a maximum duration of t_{max} and the time messages travel has to be considered as well (one unit for each request and response), the duration of the *ComputeOutgoingEdgesSecondStepTimer* is computed as follows:

  $$duration = t_{max} + 2$$

  If a cluster internal neighbor forwards the request in this time, the initiating node has to suspend the timer:

  $$duration = t_{max} + 2 + timeLeft$$

  $t_{max} + 2$ is the amount of time it can take for a node to answer the request and for the forwarding node to receive the response. $timeLeft$

is the time left for other nodes to forward the request if necessary.

ComputeOutgoingEdgesSecondStepTimer

if *this started algorithm* **then**
  add destinations to list of clusters to wait for;
  send CheckForIntersectionsRequest (broadcast);
end
else if *CheckForIntersectionsRequest exists* **then**
  compute intersections;
  send IntersectionsFoundResponse (no broadcast);
end

- **FirstRequestTimer**
  This timer is a contending timer, i.e. the node competes with other cluster internal neighbors for the right to forward the *FirstRequest*. It is started dependent on the node's distance to the destination cluster's middle:

$$duration = \frac{distanceToDestinationClusterMiddle}{1.5 * r} * t_{max}$$

  $1.5 * r$ is the maximum distance of a node to the destination cluster's middle. The maximum range of a node is r. The maximum distance of a node to its cluster's middle is $0.5 * r$, i.e. when a node just reaches a destination cluster the maximum distance is $1.5 * r$. This computation ensures that the node nearest to the destination cluster's middle answers first. If a cluster internal neighbor forwards the request in this time, the node has to suspend the timer:

$$duration = t_{max} + 2 + timeLeft$$

  $t_{max} + 2$ is the amount of time it can take for a node to answer the request and for the forwarding node to receive the response. $timeLeft$ is the time left for this node to forward the request if necessary.

FirstRequestTimer
send FirstRequest (broadcast);

- **FirstResponseTimer**
  This timer is a contending timer, i.e. the node competes with other nodes in the cluster for the right to answer the *FirstRequest*. This timer is started dependent on the node's distance to its own cluster's middle:

$$duration = \frac{distanceToOwnClusterMiddle}{0.5 * r} * t_{max}$$

$0.5 * r$ is the maximum distance of a node to the cluster middle. This computation ensures that the node nearest to the cluster middle, of all reachable nodes, answers first.

FirstResponseTimer
send FirstResponse (broadcast);

4.2.4 Proofs

This subsection contains some proofs necessary for the algorithm to be applicable. Unfortunately the coexistence property is not satisfiable for overlay graphs constructed with BLLRAP. Neither could an alternative proof for the connectivity of the overlay graph be found. The general idea was to proof that redundancy and coexistence could be inherited from the physical graph. Both properties are valid in UDGs.

4.2.4.1 Redundancy Property

To proof the redundancy property after Frey for overlay graphs of UDGs constructed with BLLRAP algorithm, another version of the redundancy property is used. This version was introduced by Sumesh [21] and is defined as follows:

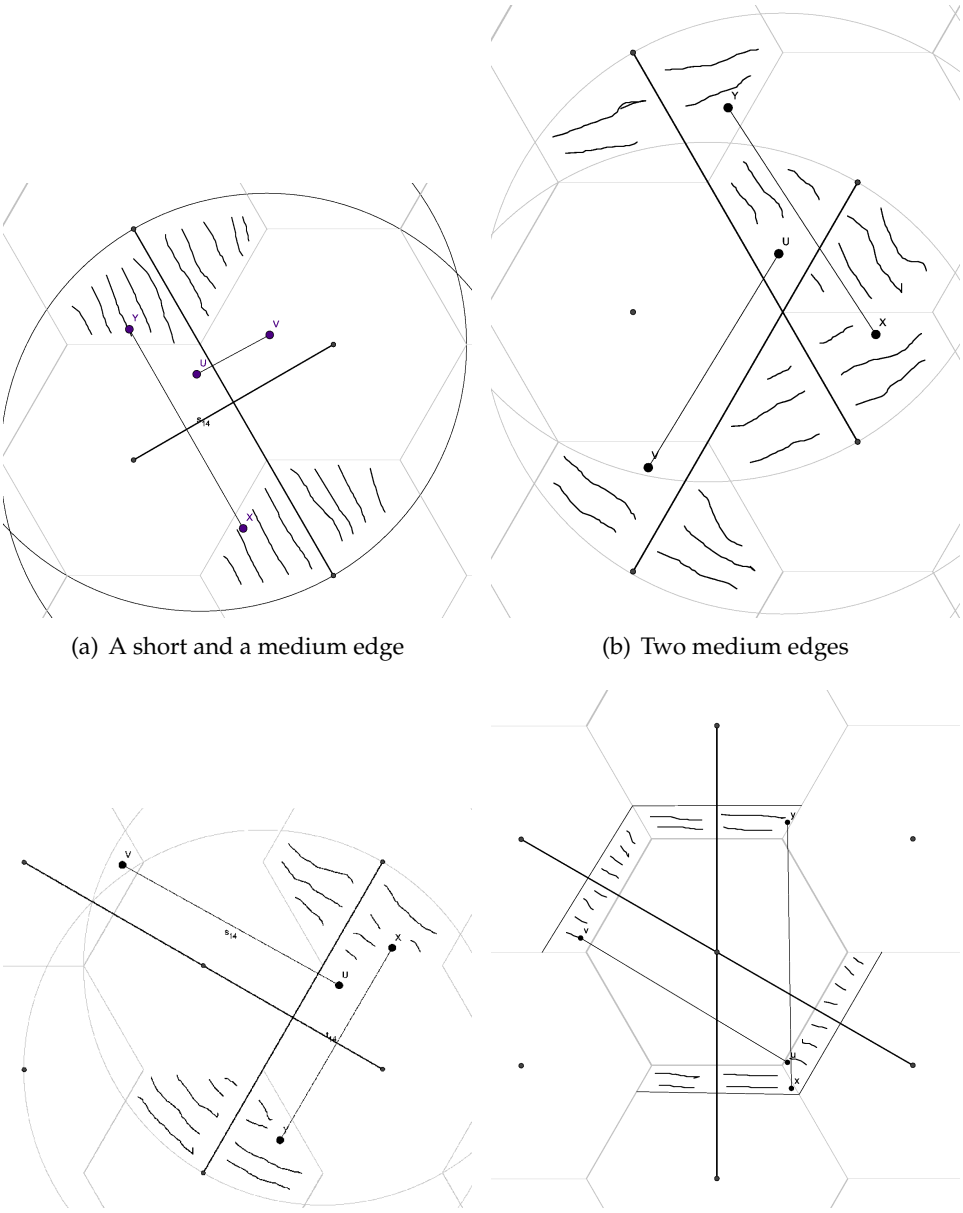
Definition 3. *If two edges in the overlay graph intersect, one of the following must occur: (a) two radio edges intersect in the UDG (b) the two radio edges in the UDG do not intersect, but the location of the four mobile nodes is such that the overlay edge intersection occurs. In either case, there must be at least one node which is directly connected to the remaining three nodes in the UDG.*

Since the redundancy property after Sumesh was only proven for square grids, a proof for hexagon grids follows:

Lemma 1. *The redundancy property defined by Sumesh in [21] is applicable to heaxagon grids as well.*

Proof. To show:
Every intersection in an overlay graph of an UDG constructed by node aggregation, is caused by three nodes where at least one is connected to the other three.

Analogous to the proof for square grids in [21] only edges that do not intersect in the original graph need to be considered. Figure 4.1 shows all possible intersections. Hatched areas indicate that a node has to lie in this area for an edge to be able to form. In case a) node u (endpoint of short



(a) A short and a medium edge

(b) Two medium edges

(c) A medium and a long edge

(d) Two long edges

Figure 4.1: Intersections between edges in an UDG overlay graph

edge) has to lie right/left from the edge xy. This leads to u always lying between x and y and therefore $|xy| \leq r \wedge |ux| \leq r \wedge |uy| \leq r$. Node u is always connected to all three other nodes that form the intersection and the property holds for short and medium edges. The cases b), c) and d) can be treated accordingly. □

Theorem 1. *When constructing an overlay graph from an UDG by using BLLRAP, the overlay graph satisfies the redundancy property after Frey.*

Proof. From the redundancy property as defined by Sumesh in [21] it follows, that at least one node of the four nodes involved in creating an intersection in the overlay graph of an UDG has edges to all three other nodes. Since the edges between the four nodes lead to an intersection in the overlay graph all four nodes have to be in different clusters. If an edge between two nodes in different clusters exists, then an edge in the overlay graph connecting the two clusters the nodes belong to exists as well. Therefore if there is an intersection in the overlay graph one of the clusters has to have edges to all three other clusters involved and the redundancy property defined by Frey (see Section 3.10) is satisfied by the overlay graph □

4.2.4.2 Coexistence Property

The coexistence property is not satisfied by the overlay graph constructed when applying BLLRAP. The constellation in Figure 4.2 shows a contradiction. Although a node lies in cluster D, which therefore exists, it has no edge to cluster C, since node w is not reachable by node v.

4.2.4.3 Connectivity

Since the algorithm does not satisfy the coexistence property which is necessary in the original algorithm to proof the continued connectivity, an alternate proof for connectivity was sought. For that purpose a case study of all possible constellations of edges in a hexagon raster that form a triangle (possible alternate paths messages can take) were considered. Figures 4.3(a) (case 1), 4.3(b) (case 2), 4.3(e) (case 3), 4.4 (case 4), and 4.3(c), 4.3(d) (case 5 a + b) show those constellations.

The proofs are conducted by contradiction (analogously to the original proof in [16]). It is assumed that the graph is disconnected after the execution of the algorithm. The goal is to show that this could not have happened. The edge uw is always the edge that still exists after the execution of the algorithm, whereas wv and uv are the removed edges that led to the disconnection. In case 1 (4.3(a)) the edges vw or uw cannot be added again only if the edges wa or ub were kept. If that were the case however wa/ub would have been removed due to intersections so that at least one of the edges wv and uv would still have to exist. The cases 3 and 4c can be

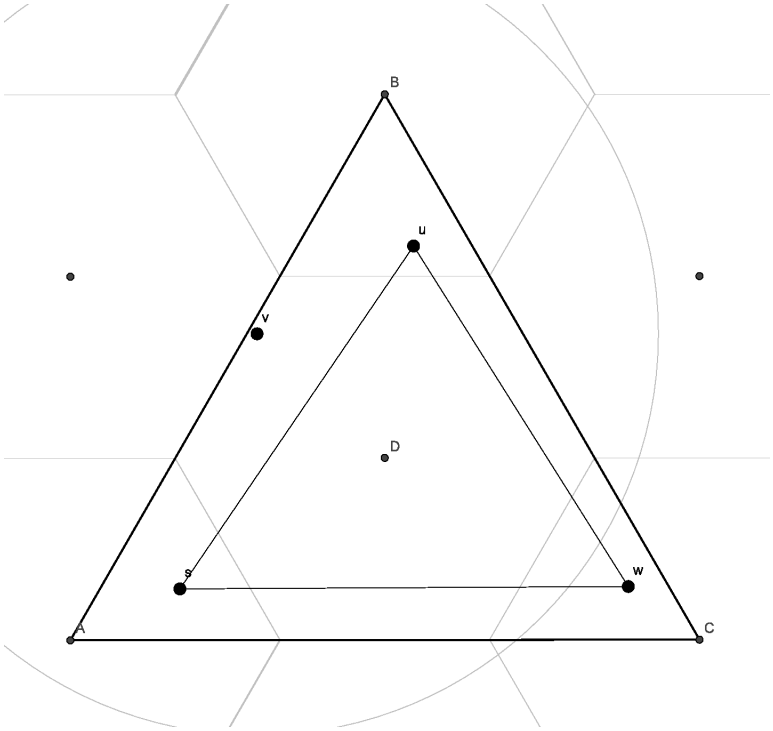
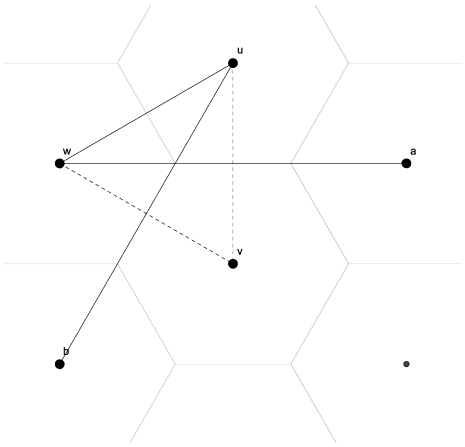
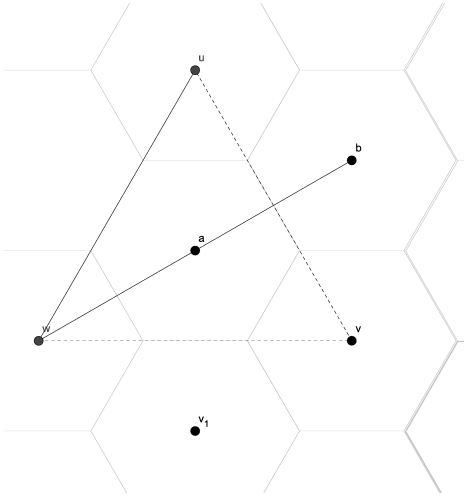


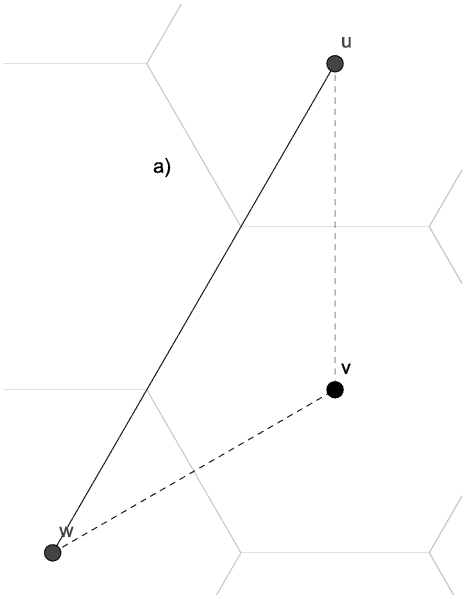
Figure 4.2: A contra-dictionary example for the coexistence property in overlay graphs of UDG



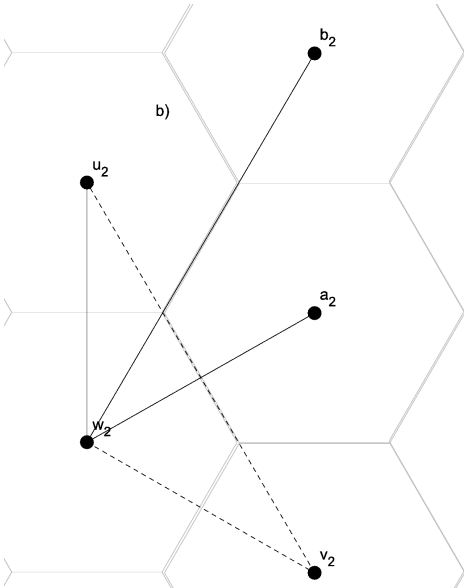
(a) Three short edges



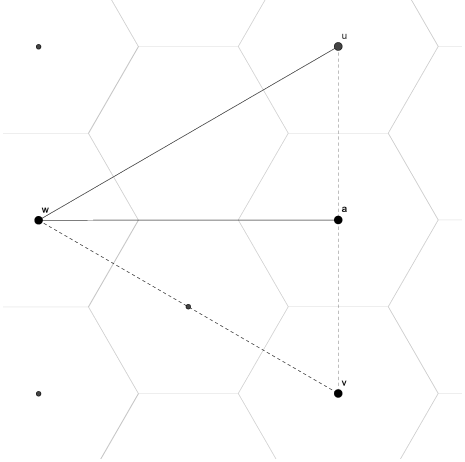
(b) Three medium edges



(c) Two short edges and one medium edge (a)

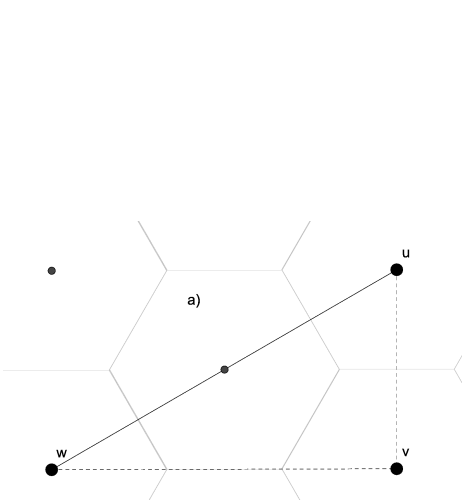


(d) Two short edges and one medium edge (b)

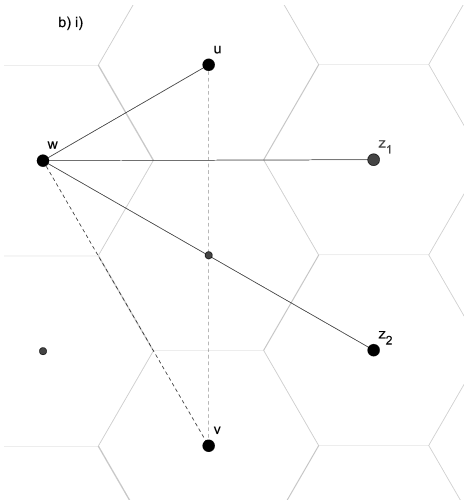


(e) Three long edges

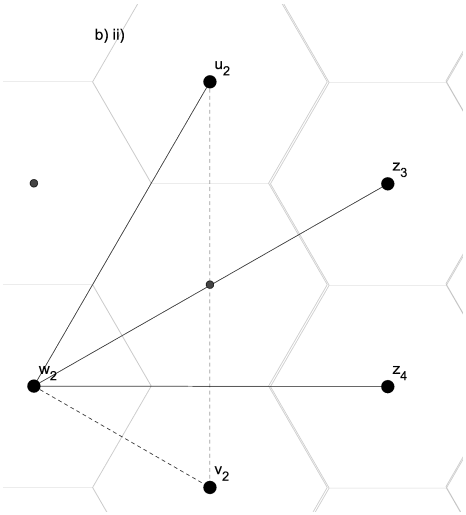
Figure 4.3: Constellations of of possible detours (part 1)



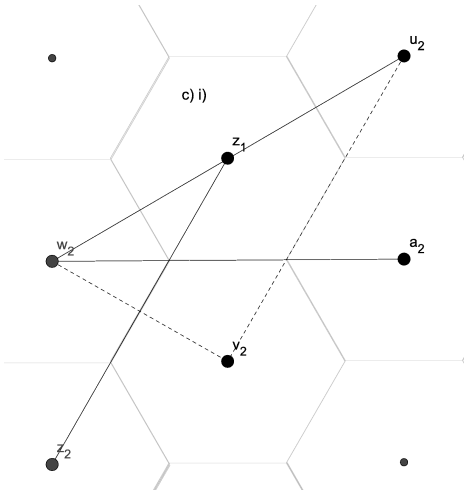
(a) One edge each (a)



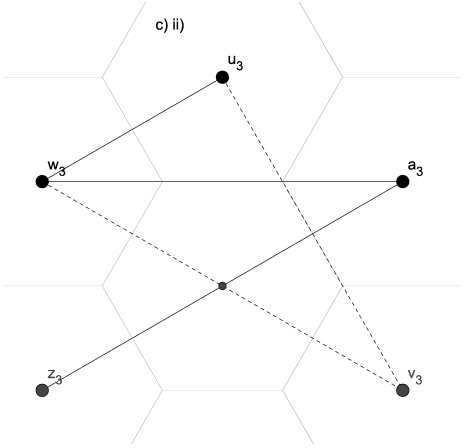
(b) One edge each b) i)



(c) One edge each b) ii)



(d) One edge each c) i)



(e) One edge each c) ii)

Figure 4.4: Constellations of of possible detours (part 2)

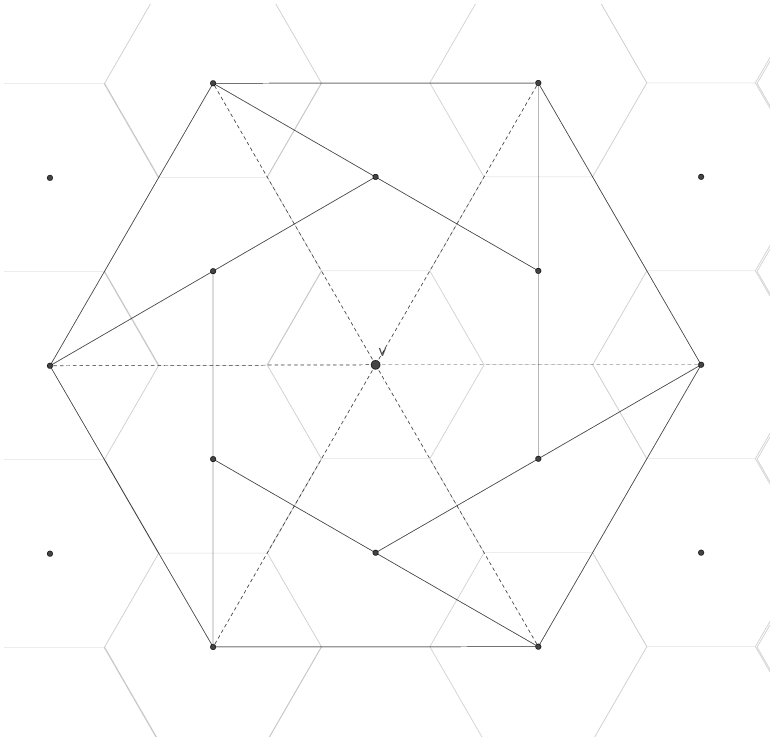


Figure 4.5: A contradiction to the assumption that an overlay graph always remains connected after LLRAP

argued in the same way. Cases 5 i) and 4a do not have any possible intersections that could lead to the removal of the edges uv and uw. The only possible intersection in case 5 ii) leads to the same constellation as in case 1) and terminates there. Cases 4b and 2 are the problematic cases which in the end lead to the conclusion that connectivity can not be proven this way. There are cases in which no edge exists that might prevent the edges uv and wv from being removed or more precisely the effort to proof that such an edge exists ends in an endless loop of detours. Figure 4.5 shows a contradiction to the assumption that an overlay graph remains connected after the algorithm. All edges from cluster V to the surrounding clusters have been removed due to intersections so that V lies now disconnected from the rest of the graph.

4.3 Beaconless Cluster Based Planarization (BCBP)

The BCBP algorithm is based upon the algorithm proposed by Frey as described in Section 3.9 and uses the same denomination for the result sets. In contrast to the original algorithm, this algorithm directly computes the result sets on cluster basis. This section contains a description of the de-

veloped BCBP algorithm as well as lists of all used messages and timers. Furthermore a proof for the correctness of the algorithm is given and optimization opportunities are described.

4.3.1 Description

The BCBP algorithm utilizes a number of messages which are explained in detail in Subsection 4.3.3 as well as several different kinds of timers (Section 4.3.4). When a node needs to know all outgoing edges of the planar overlay graph it kicks off computing of those edges through sending an *determineOutgoingEdgesRequest* to its own cluster. All nodes in this cluster then start a *DetermineClusterHeadTimer*. The node whose timer first terminates is the clusterhead of this cluster and responsible for the further execution of the algorithm. The algorithm consists of five main steps:

1. compute all outgoing Gabriel Graph edges of the cluster
2. add all short and medium edges to $E(C)$
3. check for all long edges whether a cluster in the middle of a long edge can be reached by v and add the edge if need be to $R(C)$
4. ask all endpoints of long edges (in cluster E) that are not already removed if they can reach the cluster in the middle (D) and either add CD to $I_s(v)$ and CE to $R(C)$, or add CE to $E(C)$
5. ask all reachable directly neighboring clusters, when C can't reach the cluster opposite of it (D), if a long edge over C exists to that cluster and add the edge CD to $I_r(v)$ if need be

In the first step the clusterhead initiates the determination of all Gabriel Graph edges of the cluster. For this the BFP algorithm, described in Section 3.8, is used. In the second step all short and medium edges are directly added to $E(C)$. If there are long edges, after that the clusterhead sends an *ExistsAnyEdgeRequest* to all directly adjacent clusters. All nodes in those clusters which receive the request start an *EdgeExistTimer*. When the timer terminates the node sends an *EdgeExistsResponse* to the clusterhead and all other nodes stop their timers. In case there are open directly neighboring clusters, the clusterhead then forwards the *ExistsAnyEdgeRequest* to its cluster internal neighbors. The cluster internal neighbors then start a *DetermineNearestNeighborTimer* for the right to forward the request. When one of the timers terminates, the node sends an *ExistsAnyEdgeRequest* to the specific cluster. If other nodes started a timer as well, it is suspended. The information about existing edges is send back to the clusterhead via a *ReachableClustersResponse*. When no node can reach any of the open clusters anymore, the clusterhead checks which of the long edges have to be removed because they cause irregular intersections and adds them to $R(C)$.

In the fourth step the clusterhead sends an *CheckForIrregularIntersectionsRequest* to the endpoints of all open long edges, routing over intermediate nodes if necessary. This causes a stripped version of the third step to be executed in the destination clusters. Instead of searching for edges to all directly adjacent clusters only the cluster in the middle is checked. After the step is executed feedback in form of an *IntersectionFoundResponse* is send to the clusterhead. If an intersection was found the long edge is added to $R(C)$ and the edge from the cluster to the middle cluster is added to $I_s(v)$, otherwise the long edge is added to $E(C)$. In the fifth step the clusterhead computes which directly adjacent clusters can't be reached. If the cluster opposite of them can be reached, it sends an *ExistsLongEdgeOverMeRequest* to those clusters (if necessary over an intermediate node). Those clusters then use a modified version of the BFP algorithm (see Subsection 4.3.2 to determine whether they have a Gabriel Graph edge to the opposite cluster. After that is finished they send an *LongEdgeExistsResponse* back to the clusterhead. If such a long edge exists, the edge from the cluster to the reachable cluster is added to $I_r(v)$. Finally the clusterhead sends an *OutgoingEdgesResponse* containing all four edge result sets to the initial node.

4.3.2 Modified Beaconless Forwarder Planarization

The modified version of the BFP algorithm works on cluster basis. Only nodes that lie in a specified destination cluster start the *GGCandidateTimer*. However, every node that lies in the proximity region of an answering node and the requesting node, still adds the answering node to its protest list and sends a *ProtestResponse* if necessary. Regardless of the cluster it lies in. Thus, it is ensured that no false candidate is mistaken for a Gabriel Graph neighbor. Since all nodes in the destination cluster that overhear the *SearchForGGCandidatesRequest* start a *GGCandidateTimer*, no true Gabriel Graph neighbor in that cluster is missed.

4.3.3 Messages

This subsection contains a list of all messages used in the BCBP algorithm. For each message a brief pseudo code description on how to process it is given. A detailed diagram of the message flow can be found in appendix A.

DetermineOutgoingEdgesRequest

```
if message from this cluster then
    if do not forward message then
        start DetermineClusterHeadTimer;
        // dependent on the distance to the middle of
           the cluster
    else
        start DetermineNearestNeighborTimer;
        // dependent on the distance to the nearest
           open cluster
    end
end
```

SearchForGGCandidatesRequest

```
if message for this cluster then
    start GGCandidateTimer;
    // dependent on the distance to the requesting
       node
else
    else if message from this cluster then
        if this is clusterhead then
            if DetermineOutgoingEdgesTimer running then
                suspend timer for 3 * t_max + 10;
            end
            else if LongEdgeExistsTimer running then
                suspend timer for 3 * t_max + 10;
            end
            // another node answered at the same time
            else if requesting nodes ID < this ID then
                this is not clusterhead;
            end
        else
            if DetermineClusterHeadTimer running then
                stop timer;
            end
            if DetermineNearestNeighborTimer running then
                suspend timer for 2 * t_max + 6 + timeleft;
            end
        end
    end
end
```

GGCandidateResponse

```
if message for me then
    add responding node to list of potential edges;
else
    if GGCandidateTimer running && responding node is in proximity
    region then
        stop timer;
    end
    else if this node is in proximity region then
        add responding node to protest list;
    end
end
```

ReadyForProtestRequest

```
if protest list is not empty then
    start ProtestTimer;
    // dependent on the distance to the requesting
       node
end
```

ProtestResponse

```
if message for me then
    remove nodes from potential edges list;
else
    cross check with own protest list and remove if possible;
    if own protest list empty then
        stop timer;
    end
end
```

FoundEdgesResponse

```
if message for me then
    if not outgoing edge computation then
        add found edges to list of potential edges;
    else
        send OutgoingEdgeExistsResponse (no broadcast);
    end
end
else if message from this cluster && DetermineNearestNeighborTimer
running then
    remove found edges from list of open clusters;
    if no open clusters left then
        stop timer;
    end
end
```

CheckForIrregularIntersectionRequest

```
if message from this cluster then
    forward request;
    // message is not broadcasted
else
    send ExistsAnyEdgeRequest (broadcast);
    start ExistsAnyEdgeTimer;
end
```

IrregularIntersectionFoundResponse

```
if message needs to be forwarded then
    forward message;
else
    remove long edge from list of open long edges;
    add edge to list of outgoing edges or to list of implicit edges in
    the same direction;
    if no open long edges then
        determine clusters to check for implicit edges in the reverse
        direction;
        if clusters to check then
            send ExistsLongEdgeOverMeRequest to each cluster on
            the list (no broadcast);
            // no broadcast so forwarding is possible
        else
            finish();
        end
    end
end
```

ExistsAnyEdgeRequest

```
if message needs to be forwarded && message form this cluster then
    start DetermineNearestNeighborTimer;
    // dependent on the distance to the nearest open
       cluster
else
    if message is for me then
        start EdgeExistsTimer;
        // dependent on distance to requesting node
    end
    else if ExistsAnyEdgeTimer running then
        suspend timer for t_max + 10;
    end
    else if message from this cluster then
        if DetermineNearestNeighborTimer running then
            suspend timer for t_max + 5 + timeleft;
        end
    end
end
```

EdgeExistsResponse

```
if message is for me then
    add edge to list of reachable adjacent clusters;
end
else if message from this cluster && EdgeExistsTimer running then
    stop timer;
end
```

ReachableClustersResponse

```
if message is for me then
    add edge to list of reachable adjacent clusters;
end
else if message from this cluster && DetermineNearestNeighborTimer
running then
    remove found edges from list of open clusters;
    if no open clusters left then
        stop timer;
    end
end
```

ExistsLongEdgeOverMeRequest

```
if message from this cluster then
    forward request;
    // message is not broadcasted
else
    send ExistsOutgoingEdgeRequest (broadcast);
    start DetermineClusterHeadTimer;
    // dependent on distance to cluster middle
end
```

ExistsOutgoingEdgeRequest

```
if message from this cluster then
    if message needs to be forwarded then
        start DetermineNearestNeighborTimer;
        // dependent on distance to destination
           cluster
    else
        start DetermineClusterHeadTimer;
        // dependent on distance to cluster middle
    end
end
```

OutgoingEdgeExistsResponse

```
if message from this cluster then
    send LongEdgeExistsResponse;
end
else if message needs to be forwarded then
    forward message;
end
else
    remove cluster from list of clusters to check for implicit edges in
    the reverse direction;
    add edge to list of implicit edges in the reverse direction if
    necessary;
    if no clusters to check then
        finish();
    end
end
```

LongEdgeExistsResponse

```
if message needs to be forwarded then
    forward message;
else
    remove cluster from list of clusters to check for implicit edges in
    the reverse direction;
    add edge to list of implicit edges in the reverse direction if
    necessary;
    if no clusters to check then
        finish();
    end
end
```

OutgoingEdgesResponse

```
extract outgoing edges from message;
```

4.3.4 Timers

This subsection contains a list of all timers used in the BCBP algorithm. For each timer a brief pseudo code description of what happens when the timer fires is given as well as information about how their duration, and if necessary suspension, is determined. All timers are dependent on a variable called t_{max}, which is the maximum amount of time a timer can be initially

set to when contending for the right to answer a request.

- **DetermineClusterHeadTimer**
  This timer is a contending timer, i.e. the node competes with other nodes in the cluster for the right to send the *SearchForGGCandiates-Request*. This timer is started dependent on the node's distance to its own cluster's middle:

$$duration = \frac{distanceToOwnClusterMiddle}{0.5 * r} * t_{max}$$

  $0.5 * r$ is the maximum distance of a node to the cluster middle. This computation ensures that the node nearest to the cluster middle, of all reachable nodes, answers first.

DetermineClusterHeadTimer
send SearchForGGCandidatesRequest (broadcast);

- **DetermineNearestNeighborTimer**
  This timer is a contending timer, i.e. the node competes with other cluster internal neighbors for the right to forward a request. It is started dependent on the node's distance to the (nearest) destination cluster's middle:

$$duration = \frac{distanceToDestinationClusterMiddle}{1.5 * r} * t_{max}$$

  $1.5 * r$ is the maximum distance of a node to the destination cluster's middle. The maximum range of a node is r. The maximum distance of a node to its cluster's middle is $0.5 * r$, i.e. when a node just reaches a destination cluster the maximum distance is $1.5 * r$. This computation ensures that the node nearest to the (for its nearest) destination cluster's middle answers first. If a cluster internal neighbor forwards the request in this time, the node has to suspend the timer depending on the request to be forwarded:

$$duration = 2 * t_{max} + 5 + timeLeft$$

  for a *SearchForGGCandidatesRequest* and

$$duration = t_{max} + 3 + timeLeft$$

  for an *ExistsAnyEdgeRequest*. $2 * t_{max} + 5$ is the amount of time it can take for the forwarding node to determine all its Gabriel Graph neighbors (2 requests, 3 responses and 2 timers with a duration of at most t_{max}). $t_{max} + 3$ is the amount of time it can take for the forwarding

node to determine all its directly neighboring clusters (1 requests, 2 responses and 1 timer with a duration of at most t_{max}). $timeLeft$ is the time left for this node to forward the request if necessary.

DetermineNearestNeighborTimer

if *part of GG computation* **then**
  send SearchForGGCandidatesRequest (broadcast);
  start SearchForGGCandidatesTimer;
else
  send ExistsAnyEdgeRequest (broadcast);
  start ExistsAnyEdgeTimer;
end

- **DetermineOutgoingEdgesTimer**
  This timer is a structural timer, i.e. it is used to structure the course of events. It is started when a node starts the computation of all outgoing cluster edges that can be reached over an intermediate node. Since the timer used to determine the node to forward the request (*DetermineNearestNeighborTimer*), has a maximum duration of t_{max} and the time messages travel has to be considered as well (one unit for each request and response), the duration of the *DetermineOutgoingEdgesTimer* is computed as follows:

$$duration = t_{max} + 2$$

If a cluster internal neighbor forwards the request in this time, the initiating node has to suspend the timer:

$$duration = 3 * t_{max} + 6 + timeLeft$$

$3 * t_{max} + 6$ is the amount of time it can take for a node to win the contest to forward the request and to determine all its Gabriel Graph neighbors (3 requests, 3 responses and 3 timers with a duration of at most t_{max}). $timeLeft$ is the time left for other nodes to forward the request if necessary.

DetermineOutgoingEdgesTimer

```
if any potential edges exist then
    forall the potential edges do
        switch edge type do
            case short
            case medium
                add edge to list of outgoing edges;
            end
            case long
                add edge to list of open long edges;
            end
        endsw
    end
    send ExistsAnyEdgeRequest (broadcast);
    start ExistsAnyEdgeTimer;
else
    finish();
end
```

- **SearchForGGCandidatesTimer**
  This timer is a structural timer, i.e. it is used to structure the course of events. It is started when a node starts the computation of its Gabriel Graph neighbors. Since the timer used to determine whether a node is a Gariel Graph neighbor (*GGCandidateTimer*), has a maximum duration of t_{max} and the time messages travel has to be considered as well (one unit for each request and response), the duration of the *GGCandidateTimer* is computed as follows:

$$duration = t_{max} + 2$$

SearchForGGCandidatesTimer

```
if any potential edges then
    send ReadyForProtestRequest (broadcast);
    start GGCandidateProtestTimer;
end
```

- **GGCandidateTimer**
  This timer is a contending timer, i.e. the node competes with other nodes in the cluster for the right to answer the *SearchForGGCandidatesRequest*. This timer is started dependent on the node's distance to the requesting node:

$$duration = \frac{distanceToRequestingNode}{r} * t_{max}$$

r is the maximum distance between two nodes in an UDG. This computation ensures that the node nearest to the requesting node, of all reachable nodes, answers first.

GGCandidateTimer
send GGCandidateResponse (broadcast);

- **GGCandidateProtestTimer**
  This timer is a structural timer, i.e. it is used to structure the course of events. It is started when a node starts the protest phase of the computation of its Gabriel Graph neighbors. Since the timer used to determine whether a node will send a *ProtestResponse* (*ProtestTimer*), has a maximum duration of t_{max} and the time messages travel has to be considered as well (one unit for each request and response), the duration of the *GGCandidateTimer* is computed as follows:

$$duration = t_{max} + 2$$

GGCandidateProtestTimer

```
switch type of timer do
    case general computation
        if any open clusters then
            send DetermineOutgoingEdgeRequest for forwarding
            (broadcast);
            start DetermineOutgoingEdgesTimer;
        else
            computeEdges();
        end
    end
    case forwarding computation
        if edge found then
            send FoundEdgesResponse (broadcast);
        end
    end
    case exists outgoing edge general computation
        if edge found then
            send OutgoingEdgeExistsResponse (no broadcast);
        else
            send ExistsOutgoingEdgeRequest for forwarding
            (broadcast);
            start LongEdgeExistsTimer;
        end
    end
    case exists outgoing edge forwarding computation
        if edge found then
            send FoundEdgesResponse (no broadcast);
        end
    end
endsw
```

- **ProtestTimer**
  This timer is a contending timer, i.e. the node competes with other nodes in the cluster for the right to answer the *ReadyForProtestRequest*. This timer is started dependent on the node's distance to the requesting node:

$$duration = \frac{distanceToRequestingNode}{r} * t_{max}$$

  r is the maximum distance between two nodes in an UDG. This computation ensures that the node nearest to the requesting node, of all reachable nodes, answers first.

ProtestTimer
send ProtestResponse (broadcast);

- **EdgeExistsTimer**
  This timer is a contending timer, i.e. the node competes with other nodes in the cluster for the right to answer the *ExistsAnyEdgeRequest*. This timer is started dependent on the node's distance to the requesting node:

$$duration = \frac{distanceToRequestingNode}{r} * t_{max}$$

  r is the maximum distance between two nodes in an UDG. This computation ensures that the node nearest to the requesting node, of all reachable nodes, answers first.

EdgeExistsTimer
send EdgeExistsResponse (broadcast);

- **LongEdgeExistsTimer**
  This timer is a structural timer, i.e. it is used to structure the course of events. It is started when a node starts the forwarding computation of a long edge Gabriel Graph neighbor. Since the timer used to determine the node to forward the request (*DetermineNearestNeighborTimer*), has a maximum duration of t_{max} and the time messages travel has to be considered as well (one unit for each request and response), the duration of the *LongEdgeExistsTimer* is computed as follows:

$$duration = t_{max} + 2$$

  If a cluster internal neighbor forwards the request in this time, the initiating node has to suspend the timer:

$$duration = 3 * t_{max} + 6 + timeLeft$$

  $3 * t_{max} + 6$ is the amount of time it can take for a node to win the contest to forward the request and to determine if it has a Gabriel Graph neighbor in the destination cluster (3 requests, 3 responses and 3 timers with a duration of at most t_{max}). $timeLeft$ is the time left for other nodes to forward the request if necessary.

LongEdgeExistsTimer
send OutgoingEdgeExistsResponse (no broadcast);

- **ExistsAnyEdgeTimer**
  This timer is a structural timer, i.e. it is used to structure the course of events. It is started when a node starts the computation of all reachable directly adjacent clusters. Since the timer used to determine whether a node will send a *EdgeExistsResponse* (*EdgeExistsTimer*), has a maximum duration of t_{max} and the time messages travel has to be considered as well (one unit for each request and response), the duration of the *ExistsAnyEdgeTimer* is computed as follows:

  $$duration = t_{max} + 2$$

  When the node is waiting for the results of forwarding the request and a cluster internal neighbor forwards the request in this time, the initiating node has to suspend the timer:

  $$duration = t_{max} + 2 + timeLeft$$

  $t_{max} + 2$ is the amount of time it can take for a node to determine whether it can reach a node in the destination clusters (1 request, 1 response and 1 timer with a duration of at most t_{max}). $timeLeft$ is the time left for other nodes to forward the request if necessary.

ExistsAnyEdgeTimer

```
if general computation then
    if any clusters open then
        send ExistsAnyEdgeRequest for forwarding(broadcast);
        start ExistsAnyEdgeTimer;
    else
        computeIrregularIntersections();
    end
end
else if waiting on forwarding during general computation then
    computeIrregularIntersections();
end
else if general computation during forwarding computation then
    if any clusters open then
        send ExistsAnyEdgeRequest for forwarding(broadcast);
        start ExistsAnyEdgeTimer;
    else
        send IrregularIntersectionFoundResponse (no broadcast);
    end
end
else if waiting on forwarding during forwarding computation then
    send IrregularIntersectionFoundResponse (no broadcast);
end
else if forwarding computation then
    send ReachableClustersResponse (broadcast);
end
```

computeEdges()

```
if any potential edges exist then
    forall the potential edges do
        switch edge type do
            case short
            case medium
                add edge to list of outgoing edges;
            end
            case long
                add edge to list of open long edges;
            end
        endsw
    end
    send ExistsAnyEdgeRequest (broadcast);
    start ExistsAnyEdgeTimer;
else
    finish();
end
```

computeIrregularIntersections()

```
forall the open long edges do
    if forms irregular intersection then
        remove edge from list of open long edges;
    end
end
if any open long edges then
    send CheckForIrregularIntersectionsRequest to each open cluster
    (no broadcast);
    // no broadcast so forwarding is possible
else
    determine clusters to check for implicit edges in the reverse
    direction;
    if clusters to check then
        send ExistsLongEdgeOverMeRequest to each cluster on the
        list (no broadcast);
        // no broadcast so forwarding is possible
    else
        finish();
    end
end
```

finish()

if *this started algorithm* **then**
  finished;
else
  send OutgoingEdgesResponse to initial node (no broadcast);
end

4.3.5 Proofs

This subsection contains the proof of correctness for the algorithm described above. The algorithm will be considered correct if the result is the same as the one of the original algorithm. Let V be an arbitrary but finite set of nodes in the plane. Let $U(V)$ be the corresponding unit disk graph. Let C be an arbitrary but fixed cluster in $U(V)$. Let several sets of edges in the aggregated Gabriel Graph $H(G(V))$ be defined as follows:
Computed with the original algorithm after Frey:

- E_o set of outgoing edges of C
- I_{so} set of implicit edges in the same direction of C
- I_{ro} set of implicit edges in the reverse direction of C
- R_o set of removed long edges of C

Computed with the BCBP algorithm described in this section:

- E_b set of outgoing edges of C
- I_{sb} set of implicit edges in the same direction of C
- I_{rb} set of implicit edges in the reverse direction of C
- R_b set of removed long edges of C

Gabriel Graph edges are always unambiguous, i.e. every algorithm will always extract the same Gabriel Graph edges from an original graph.

Lemma 2. *To show:*
An edge $CD \in H(G(V))$ is in the set E_b if and only if it is in the set E_o.
An edge $CD \in H(G(V))$ is in the set E_o if and only if it is in the set E_b.

Proof. An edge $CD \in H(G(V))$ is in the set E_b if and only if it is in the set E_o.

Let CD be an edge of the set E_b. CD can be a short, medium or long edge. If it is a short or medium edge the same edge is in E_o since those edges are in both algorithms always added to $E(C)$ without needing to fulfill any further conditions besides being a Gabriel Graph edge. If it is a long edge it was only added to E_b if no irregular intersections were found in the steps three and four of the algorithm. Step three eliminates all long edges that the cluster itself can confirm as being part of an irregular intersection, whereas step four checks for the remaining long edges whether an irregular intersection might exist from the other side. If it had been removed either the initiating cluster or the cluster of the endpoint of the long edge would have to have an edge to the middle cluster. In this case the edge would have been removed from E_o as well.

Now we show: An edge $CD \in H(G(V))$ is in the set E_o if and only if it is in the set E_b.

Let CD be an edge of the set E_o. CD can be a short, medium or long edge. If it is a short or medium edge the same edge is in E_b since those edges are in both algorithms always added to $E(C)$ without needing to fulfill any further conditions besides being a Gabriel Graph edge. If it is a long edge it was only added to E_o if no irregular intersections were found in the first step of the original algorithm. This step checks whether a cluster D, that lies between C and E (endpoints of long edge), exists and if it can be reached by either of the endpoints. If this is the case an irregular intersection exists. If it had been removed either C or E had to have an edge to D. In this case the edge would have been removed from E_b as well. □

Lemma 3. *To show:*
An edge $CD \in H(G(V))$ is in the set I_{sb} if and only if it is in the set I_{so}.
An edge $CD \in H(G(V))$ is in the set I_{so} if and only if it is in the set I_{sb}.

Proof. An edge $CD \in H(G(V))$ is in the set I_{sb} if and only if it is in the set I_{so}.

Let CD be an edge of the set I_{sb}. For the edge to be added to this set, there has to exist another cluster E to which C has the long edge CE, and which in turn has a short edge to D (ED), while C has no direct connection to D. Otherwise step 4 of the BCBP algorithm would not have been executed and the edge therefore not been added. For this constellation of clusters the original algorithm would have done the same. In the first step it is checked whether a cluster D exists between C and E that might be reached by either of the clusters. If this is the case and C can't reach D the edge CD is added to I_{so}.

Now we show: An edge $CD \in H(G(V))$ is in the set I_{so} if and only if it is in the set I_{sb}.

Let CD be an edge of the set I_{so}. For the edge to be added to this set, there has to exist another cluster E to which C has the long edge CE, and which in turn has a short edge to D (ED), while C has no direct connection to D. Otherwise the edge would not have been considered for addition to this set, in the first step of the algorithm. For this constellation of clusters the BCBP algorithm would have done the same. In step 4 it is checked for all remaining long edges whether the end points of those edges can reach the cluster in the middle (D). If this is the case the edge CD is added to I_{sb}. □

Lemma 4. *To show:*
An edge $CD \in H(G(V))$ is in the set I_{rb} if and only if it is in the set I_{ro}.
An edge $CD \in H(G(V))$ is in the set I_{ro} if and only if it is in the set I_{rb}.

Proof. An edge $CD \in H(G(V))$ is in the set I_{rb} if and only if it is in the set I_{ro}.

Let CD be an edge of the set I_{rb}. For the edge to be added to this set, there has to exist another cluster E to which C has the short edge CE, and which in turn has a long edge to D (ED), while C has no direct connection to D. Otherwise step 5 of the BCBP algorithm would not have been executed and the edge therefore not been added. For this constellation of clusters the original algorithm would have done the same. In the second step it is checked for all removed long edges CE whether a cluster D exists between C and E that cannot be reached by C. If this is the case, the edge CD is added to I_{ro}.

Now we show: An edge $CD \in H(G(V))$ is in the set I_{ro} if and only if it is in the set I_{rb}.

Let CD be an edge of the set I_{ro}. For the edge to be added to this set, there has to exist another cluster E to which C has the short edge CE, and which in turn has a long edge to D (ED), while C has no direct connection to D. Otherwise the edge would not have been considered for addition to this set, in the second step of the algorithm. For this constellation of clusters the BCBP algorithm would have done the same. In step 5 it is checked for all clusters D opposite of those C can't reach, whether they have a the long edge ED. If this is the case the edge CD is added to I_{rb}. □

Lemma 5. *To show:*
An edge $CD \in H(G(V))$ is in the set R_b if and only if it is in the set R_o.
An edge $CD \in H(G(V))$ is in the set R_r if and only if it is in the set R_b.

Proof. An edge $CD \in H(G(V))$ is in the set R_b if and only if it is in the set R_o.

Let CD be an edge of the set R_b. For the edge to be added to this set, it had to have been removed either in step 3 or step 4 of the BCBP algorithm. In step 3 it would have been removed because C has a short edge to cluster E, which lies in the middle of CD, as well. In step 4, D would have to have had the edge DE, for the edge CD to be removed. In summary: either C or D had to have had another edge to E (or both). In this case the original algorithm would have added the edge CD to R_o as well, in the first step.

Now we show: An edge $CD \in H(G(V))$ is in the set R_r if and only if it is in the set R_b.

Let CD be an edge of the set R_o. For the edge to be added to this set, either C or E had to have had another edge to a cluster E that lies in the middle of the edge CD. If C had another edge to cluster E the edge CD would have been removed in step 3 of the BCBP algorithm. If D had another edge to cluster E the edge CD would have been removed in step 4 of the BCBP algorithm. In both cases CD would have been added to R_b. □

Theorem 2. *The results of the BCBP algorithm and the beaconless version of that algorithm are the same.*

Proof. From lemmas 2, 3, 4 and 5 it follows that all four result sets are the same for both algorithms. Therefore the overall end result is the same as well. □

Concerning termination it can be said, that the algorithm always terminates. The initiating node either waits for a timer with a finite duration to expire or for an answer from a specified number of clusters, to start the next step of the computation. Since answers are sent when a timer with a finite duration expires, every step of the algorithm terminates. Termination of the whole algorithm follows from that.

4.3.6 Optimizations

This section contains optimizations for the BCBP algorithm.

4.3.6.1 Combining of messages

For the computation of irregular intersections it is necessary to know, whether a cluster has any edge (not necessarily a Gabriel Graph edge) to neighboring clusters. In the algorithm described in Section 4.3.1 the computation of the Gabriel Graph edges and the computation whether any edge exists is done in two different steps. Those steps can be combined. If a cluster receives a *GGCandidateResponse* that automatically means it has an edge to that cluster. Independent of the fact whether the Gariel graph candidate is a true or a false candidate. Nodes who usually would not answer due to another node answering first directly send an *EdgeExistsResponse* if no other node of their cluster has already done so. In the worst case scenario (every directly neighboring cluster can only be reached over a different internal cluster neighbor) this saves up to 20 messages. In the best case scenario (the clusterhead can reach all directly neighboring clusters itself) it saves 7 messages.

4.3.6.2 Gabriel Graph candidate

In the original BFP algorithm a node does not answer a *SearchForGGCandidateRequest* if another node that lies in the proximity region answers first. Since the BFP algorithm was adapted to cluster basis for the BCBP algorithm, and nodes in the requesting cluster do not participate in the selection process, this can lead to a lot of protests from nodes of the requesting cluster. To reduce the number of false candidates a Gabriel Graph candidate node could not only take the nodes into account that answered before him, but also all other nodes it ever received a request from. This reduces the message load with increasing numbers of nodes per cluster.

4.3.6.3 Protest prediction

With a rising number of nodes per cluster the probability that no node protests against Gabriel Graph candidates in not directly neighboring clusters, decreases. If a node can predict that there will be a protest against any possible Gabriel Graph candidate from a specific cluster, it does not need to send a request to that cluster. This saves the request message as well as a candidate and a protest message. Furthermore it can enable some nodes to not send any message at all during the computation of the outgoing edges. In most cases every node of the requesting cluster would send a message at least once, when no edge to a cluster can be found.
The following criterion can be used to utilize that optimization:

1. A node determines which three corners of the destination cluster are nearest to it.

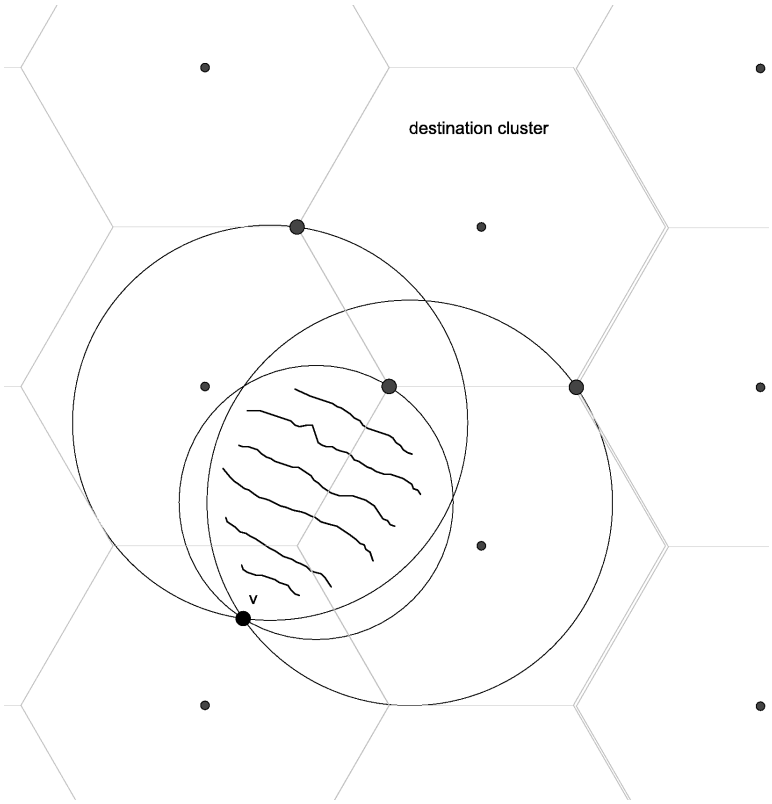


Figure 4.6: A node and the three proximity regions necessary to detect an unavoidable protest

2. The node computes the proximity regions between itself and those edges.

3. If at least one node lies in the overlapping section of all three proximity regions there is no Gabriel Graph edge to the destination cluster.

Figure 4.6 shows a node and all three proximity regions. If a node lies in the hatched zone a protest will be unavoidable.

4.3.7 Adaptation for QUDG

For the BCBP algorithm to be applicable for QUDGs the following problems have to be addressed:

1. BFP algorithm has to be adapted for QUDGs

2. overlong edges have to be considered

In [1] is explained why Gabriel Graph construction is not inheritable applicable to QUDGs. Even though all wrong edges can be removed with 2-hop

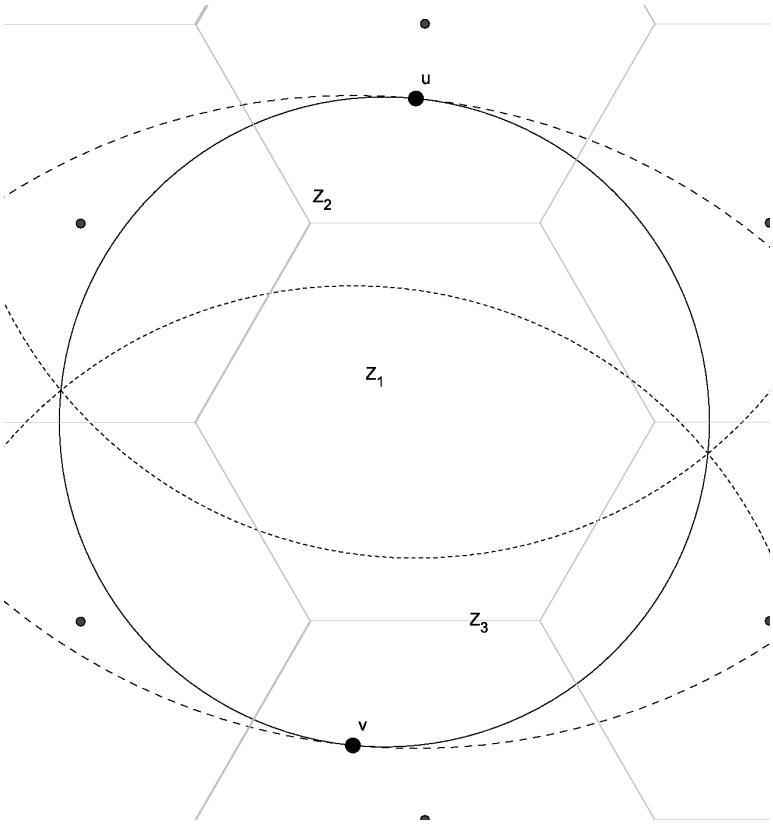


Figure 4.7: Proximity region between nodes u and v in an QUDG

information, it could lead to disconnected graphs. A proposed solution is the inclusion of virtual edges. Summarized the problems of Gabriel Graph construction for QUDGs are the following:

- a node w lying inside the proximity region between a node u and a node v could be detectable by one but not the other (every node is however always detectable by at least either u or v if $\frac{r_{max}}{r_{min}} \leq \sqrt{2}$)
- removing the edge uv without replacement can lead to disconnection

Figure 4.7 shows the possible areas inside the proximity region between u and v where another node w might be. Nodes lying in Z_1 (area that lies in the range of r_{min} of both u and v) do not have to be considered separately since they are reachable by v as well as u. For them the BFP algorithm works as it is. For nodes lying in Z_2 (area that lies in the range of r_{min} of u but only in the range of r_{max} of v) and Z_3 (area that lies in the range of r_{min} of v but only in the range of r_{max} of u) the following three situations can arise (assuming v wants to know its Gabriel Graph neighbors):

1. A node w in Z_2 is not reachable by v. It does not answer v's request.

Since v did not send a response u responds although it is not a Gabriel Graph neighbor of v.

2. A node w in Z_3 is not reachable by u. U does not hear w response and responds as well. W doesn't notice and therefore does not add u to its protest list.

3. A node w in Z_2 is not reachable by v. Node u does not answer because another node k, answered first. W is not known to any of the nodes, but could be a Gabriel Graph neighbor along a virtual edge through u.

To resolve the first situation, every node that hears a response to a request it did not hear, but lies in the proximity region, has to send a protest message to the responding node (w sends protest message to u). The responding node than protests against itself when asked, indicating the reason why (u tells v that w exists). To resolve the second situation, node u has to send a list of the candidates with the request for protests message. Every node noticing that it lies in the proximity region of one of those candidates and v, sends a protest message to v (w sends protest message to v). Both sequences of events can be optimized to avoid double responses. With those alterations in place the algorithm does not classify a wrong edge as Gabriel Graph edge. The resulting graph may however, still be disconnected. To avoid that, a third step, in which possible Gabriel Graph neighbors along virtual edges are investigated, has to be added to the BFP algorithm. Since node v knows all possible Gabriel Graph neighbors along virtual edges through protests from other nodes (or at least those in respect of which other edges had to be removed therefore causing possible disconnection), it can request of those candidates to check if they could be Gabriel Graph neighbors. Every node overhearing the request sent from v checks whether it would lie in the proximity region between v and one of those candidates and protests if necessary. The candidate nodes, if they do not already know different, send out a request as well, for every node lying in the proximity region to protest. Those protest messages are then routed over u to v. After that v knows all its Gabriel Graph neighbors. Situation three of the situations above does not need to be considered further, since in that case no disconnection is caused (no edge got removed because of w).
To integrate overlong edges into the BCBP algorithm, the only change necessary is to add them to the list of outgoing edges. Since they do not cause other irregular intersections than those with long edges, which are already covered through the existing algorithm, they do not cause any further problems. It has however, to be proven that an overlay graph based upon a Gabriel Graph of an QUDG is always planar. Analogous to the proof for UDGs the following would have to be proven:
For every possible intersection in an QUDG (as shown in Figure 3.5) it is

not possible that it still exists after the Gabriel Graph construction.

Chapter 5

Evaluation

To evaluate the efficiency of the BCBP algorithm it was simulated with the simulation framework Sinalgo [1]. This chapter contains an explanation of the setup of the simulation and the results are presented. Since the beaconless version of the LLRAP algorithm could not be proven to work correctly, this chapter will focus on the BCBP algorithm.

5.1 Simulation Setup

This section contains a basic description of the simulation set up as well as a list of all metrics used to evaluate the algorithm.

5.1.1 Setup

In general the simulation consists of a number of rounds of simulations for different node densities. The average node density is measured by

$$\frac{nodeDens * dimX * dimY}{\Pi * udr^2}$$

, which computes the number of nodes to randomly distribute onto the plane. The execution of the simulation is controlled by a perl script, which is based upon the perl script used in [17] and only differs in the number of data lines and the used command line parameter from it. The script makes

[1] Sinalgo [12] is a framework developed to simulate networks of mobile sensor nodes. Its written in JAVA and can be easily integrated into the eclispe IDE. The main purpose of Sinalgo is the simulating/testing of newly developed algorithms without having to have the otherwise required hardware at hand. A further advantage is the fact, that logical errors in the algorithm can be detected without the effort a real life test environment would take. Sinalgo offers models for all kinds of scenarios. For example the communication can be synchronous or asynchronous, the nodes can be mobile or not and the connections between nodes may be stable or not. Different algorithms are usually located in different projects. Furthermore Sinalgo offers a batch mode for simulations.

it possible to start a specific number of simulations for different node densities. Since the spanning ratio, amongst other things, is being measured, it proved to be useful to generate random connected graphs for each node density in advance. Otherwise simulation time increased greatly. For each node density a new file is generated. In each simulation round a random node in one of the middle clusters is chosen to start the algorithm. For the computation of the spanning ratio, in each non empty cluster one node is chosen to start the algorithm, since for that an overall view of the overlay graph is necessary. After the simulation finishes the results are added to the specific file. When all simulations for one node density have been executed the script computes average values and confidence intervals. Those results are written to a summary file. The simulations were executed with the following configuration parameters:

- conModel = UDG
  The connectivity model is used to decided whether the graph is modeled as an UDG or an QUDG.
- nodeDensMin = 4
  The minimal node density to be simulated.
- nodeDensMax = 14
  The maximal node density to be simulated.
- rounds = 500
  The number of simulations to be started per node density.
- dimX = 354
  The width of the plane the nodes are randomly distributed on.
- dimY = 354
  The height of the plane the nodes are randomly distributed on.
- udr = 50
  The unit disk radius of a node.
- $t_{max} = 100$
  The maximum amount of time a timer can be initially set to, when contending for the right to answer a request.

Additionally simulations for node densities 20, 25, 30, 35 and 40 were executed.

5.1.2 Metrics

The following data was collected from the simulations:

- **time [number of rounds]**
  Sinalgo simulates the time as a number of rounds. With the synchronous communication model this means sending a message takes one round and timers wait the specified number of rounds. In each round each node is active exactly once (several actions can be executed when a node is active). In the simulation the number of rounds the algorithm needed to run to completion was measured.

- **number of messages**
  Each message any node sent during the course of the simulation of the algorithm was counted.

- **number of one hop neighbors**
  The number of one hop neighbors of the initiating node, i.e. all nodes the initiating node can reach directly, was determined.

- **number of two hop neighbors**
  The number of two hop neighbors of the initiating node, i.e. all nodes the initiating node can reach over a one hop neighbor, was determined.

- **number of cluster internal neighbors**
  For the number of cluster internal neighbors each node in the same cluster as the initiating node was counted. The initiating node is not included in this number.

- **number of non used cluster internal neighbors**
  Every cluster internal neighbor which did not send any message during the course of the simulation of the algorithm was counted.

- **number of edges**
  The number of outgoing cluster edges the initiating node computed. This number includes all types of edges.

- **number of implicit edges**
  The number of implicit edges in the same or reverse direction that were computed during the course of the simulation of the algorithm.

- **number of removed long edges**
  All long edges that have been removed due to them forming an irregular intersection with another edge. This number is used to infer the number of irregular intersections.

- **spanning ratio**
  The euclidean spanning ratio of the overlay graph computed by BCBP algorithm.

- **hop spanning ratio**
  The hop spanning ratio of the overlay graph computed by BCBP algorithm.

As benchmark for the number of messages that need to be sent an estimated lower bound of sent messages of the algorithm with beaconing is used. It is assumed that all nodes in the 2-hop-neighborhood have to send at least two messages in the algorithm with beaconing. One to announce their own position and one to announce the outgoing edges they found. Since in reality more messages than that are sent due to the ongoing beaconing, it is a lower bound.

5.2 Results

This section contains the results from the simulation. The implemented version of the algorithm includes the improvements mentioned in Section 4.3.6. In general all curves are plotted with an 95% confidence interval. Since the main goal of using beaconless algorithms is the reduction of the number of sent messages, Figure 5.1 shows the numbers of sent messages of the beaconless algorithm in comparison to the estimated lower bound of sent messages of the algorithm with beaconing. As can be seen in Figure 5.1, while both algorithms roughly have to use the same number of messages for node densities four and five (about 20), as of node density six the beaconless algorithm is the more efficient one. For a node density of 14 the beaconless algorithm only has to use about 30 messages whereas the algorithm with beaconing needs about 80 messages. That is a relative difference of 37,5%. The higher the node density the greater is the difference between the two curves, i.e the higher is the relative difference of messages that need to be sent. For a node density of 40 the relative difference amounts to roughly 70%. In conclusion: the more nodes are on the plane the more efficient the beaconless algorithm is in comparison to the algorithm with beaconing.

Figure 5.2 shows the number of active nodes in comparison to all nodes in the 2-hop-neighborhood. A node is considered active if it sent at least one message during the course of the computation. For the algorithm with beaconing all nodes in the 2-hop-neighborhood would have been active. Right from the start the beaconless algorithm does not use all nodes in the 2-hop neighborhood (c.a. 6 of 10). With increasing node density the difference between the two curves grows. Both curves increase linear with the node density. This means although the average absolute number of active nodes increases with the node density from about 5 to about 35 (more nodes have to protest), the relative number of active nodes actually decreases. This is represented through the third curve (green). Whereas for node density 4 about 60% of the nodes in the two-hop neighborhood have

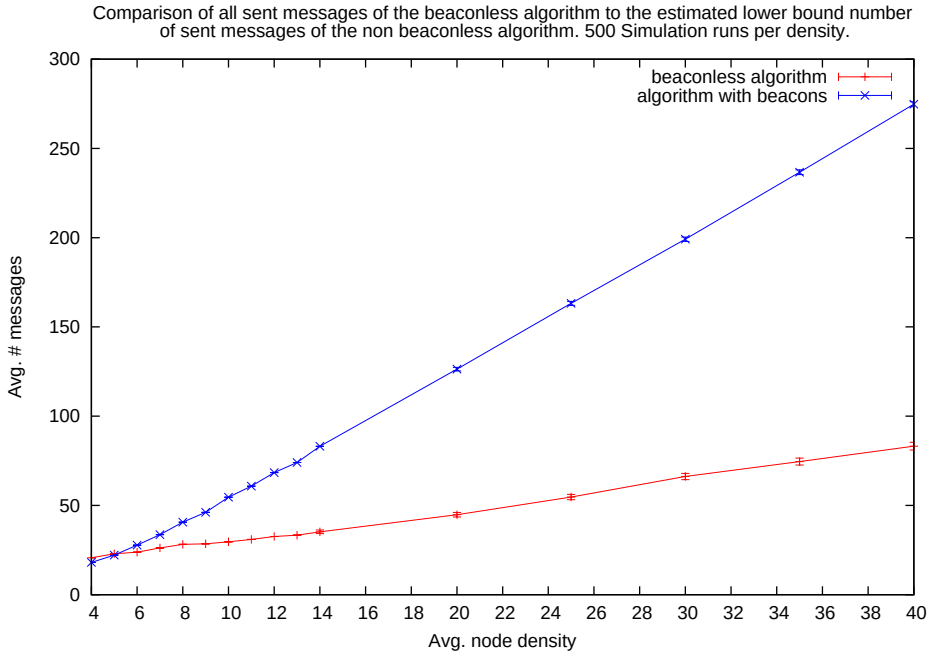


Figure 5.1: Comparison between the number of sent messages between the beaconless and the original algorithm

to send a message in the beaconless algorithm, this number decreases until for node density 40 only about 28% of the nodes in the two hop neighborhood are active. This is explained by the fact that with increasing node density there are more nodes per cluster. Since in the best case scenario only one node of each cluster has to answer, more nodes can be inactive. The relative number of nodes that are inactive decreases more slowly with increasing node density. For even higher node densities the relative number of inactive nodes may approach a constant value.

Inactive nodes in the cluster of the initiating node are not as frequent. Figure 5.3 shows the number of inactive nodes in the cluster of the initiating node in comparison to all internal cluster neighbors. As can be seen, not until node density 14 are there any noteworthy numbers of inactive internal cluster neighbors. Up to this point the number of inactive internal cluster neighbors is about 0. Even then there is not even one inactive internal cluster neighbor per simulation on average. On the other hand it can be noticed that the number of inactive internal cluster neighbors increases with the node density (from about 0 with node density 4 to about 0,25 with node density 40), so in even higher node densities the number of inactive internal cluster neighbors might increase considerably. The reason behind this phenomenon is the fact, that due to the nature of the algorithm every node in a cluster might try to reach a more distant cluster although every

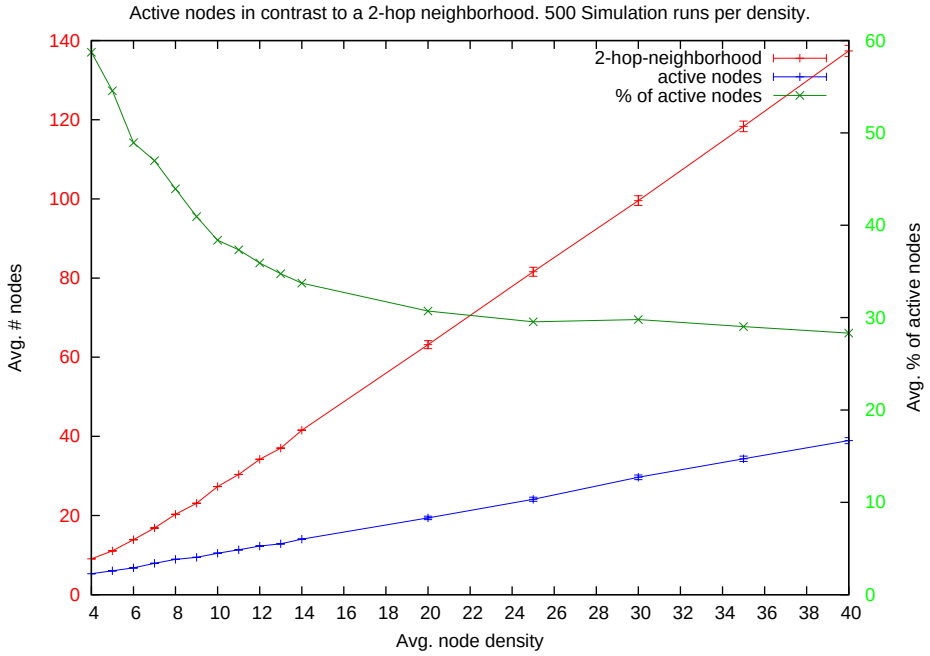


Figure 5.2: Active nodes in comparison to all possible active nodes (1-hop and 2-hop neighbors)

time the same node protests.

Figure 5.4 shows the average number of edges a cluster finds through the algorithm. The higher the node density the more the curve approaches the six edges mark. This can be explained by the fact that with higher node densities clusters tend to have only short edges, but then all possible six of them. In all simulations not even once an implicit edge in the same or an implicit edge in the reverse direction was generated. Figure 5.5 displays the number of irregular intersections (which would be responsible for those edges) per node density. As can be seen irregular intersections occur very seldom and with no kind of pattern. As of node density twenty they do not occur at all. To show that the algorithm would have detected those edges if they existed, Figure 5.6 shows an example output where an implicit edge in the same direction as well as an implicit edge in the reverse direction occurs. From node $1's$ point of view the edge between its cluster and the cluster of node 3 is an implicit edge in the same direction. From node $3's$ point of view the same edge is an implicit edge in the reverse direction. As the example shows, for an implicit edge to occur very specific conditions have to be fulfilled. The probability that this is the case decreases with increasing node density (the smaller the number of nodes on the plane the higher the chance for an implicit edge to occur). The spanning ratio is one of the key metrics used to measure the efficiency of a subgraph in compar-

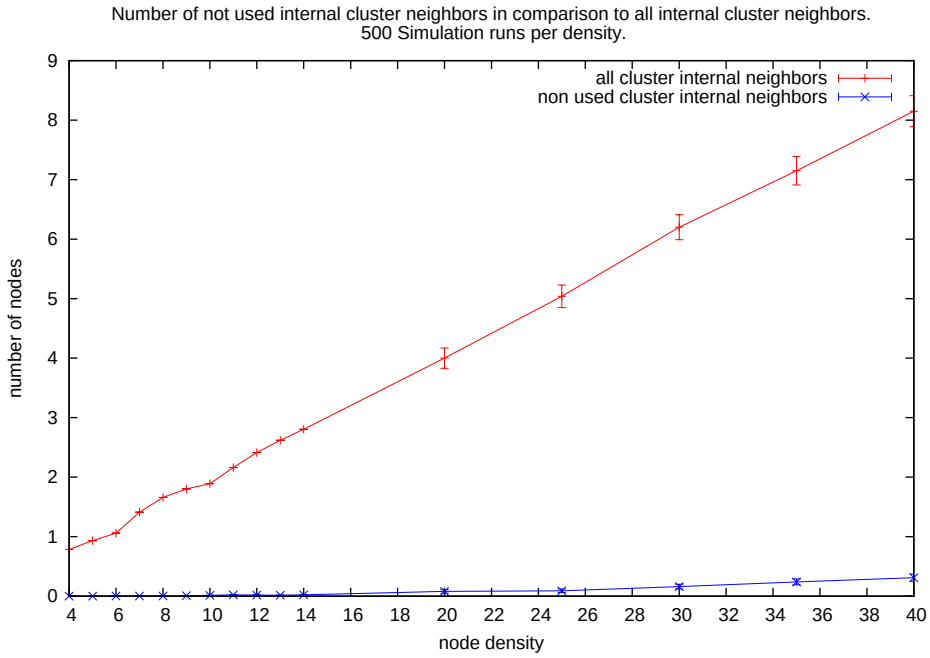


Figure 5.3: Non used internal neighbors in comparison to all internal neighbors

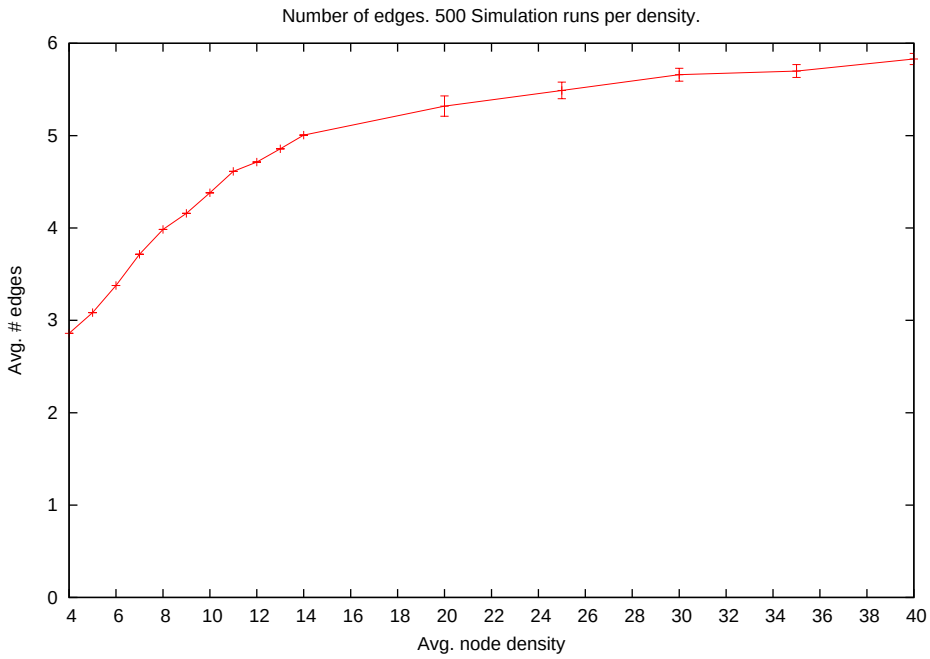


Figure 5.4: Number of edges per node density

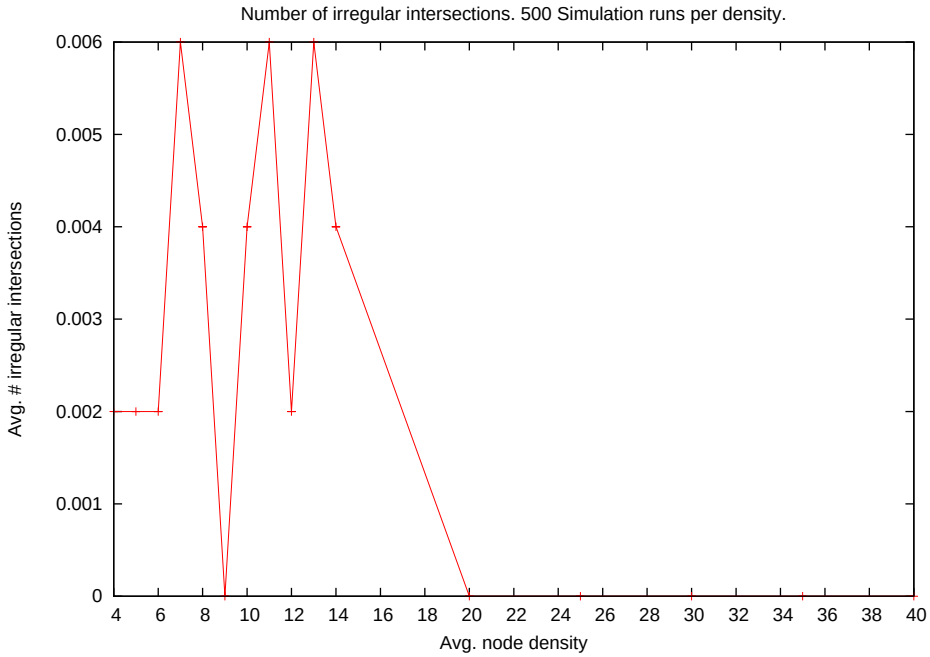


Figure 5.5: Irregular intersections per node density

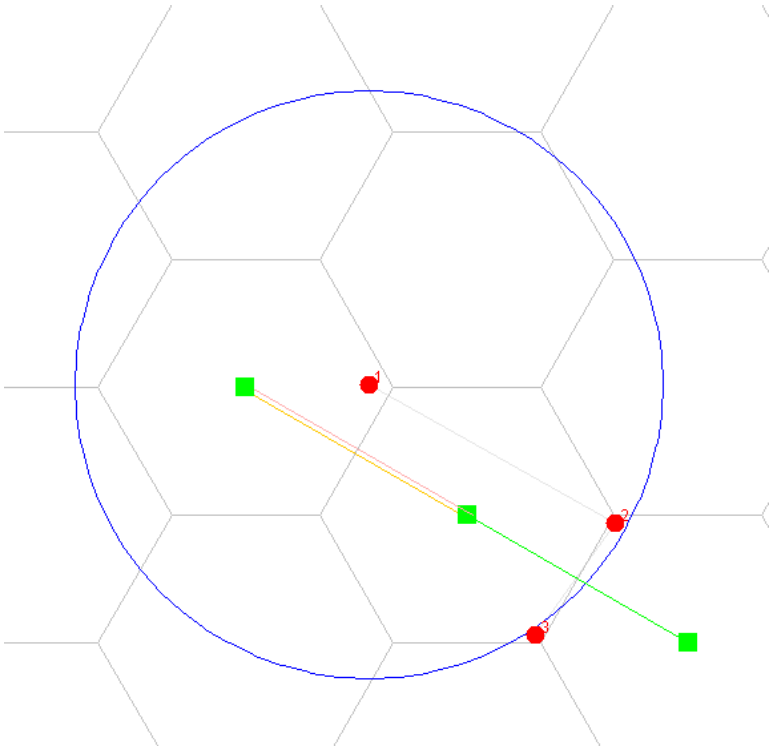


Figure 5.6: Example of computed implicit edges

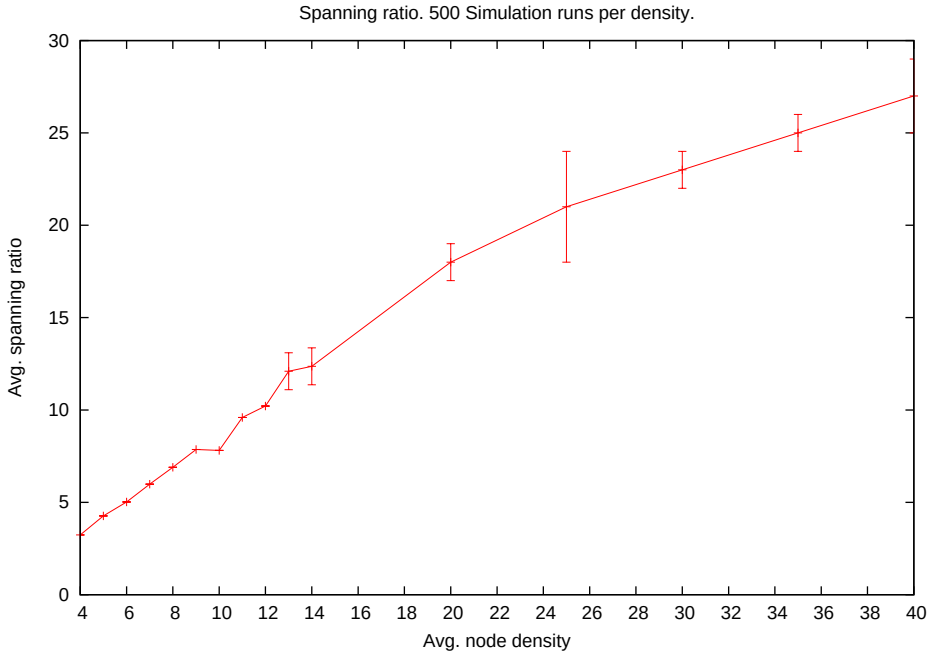


Figure 5.7: Spanning ratio with confidence intervals

ison to the original graph. Figure 5.7 shows the Euclidean spanning ratio of the beaconless algorithm for different node densities. As can be seen, the Euclidean spanning ratio increases with the node density. Whereas the Euclidean spanning ratio is about 4 for node density 4, it increases until for node density 40 the Euclidean spanning ratio amounts to about 27. It seems the algorithm is not any kind of k-spanner (with k being a constant value), since for this the spanning ratio would have to be a constant value. Although it could be that with even higher node densities the Euclidean spanning ratio approaches a constant value, since the increase is not linear. The reason why the spanning ratio has such high values is displayed in Figure 5.8. It can be the case, that two nodes lie very close together on the plane (node 101 and node 37) but that they do not have a direct connection in the overlay graph. In the example the clusterhead of node 37 already had a connection to the cluster of node 101 so that node 37 never sought any connection to the other cluster. This results in node 37 having to route a message first to its own clusterhead, which then sends it to its connection in the other cluster (node 130), which in turn sends the message to its destination (node 101). Obviously the euclidean distance of this way is much greater than the direct distance in the original graph. Since with increasing node density the probability of occurrence of such cases increases as well, the spanning ratio increases with the node density. The hop spanning ratio as displayed in Figure 5.9 increases as well with the

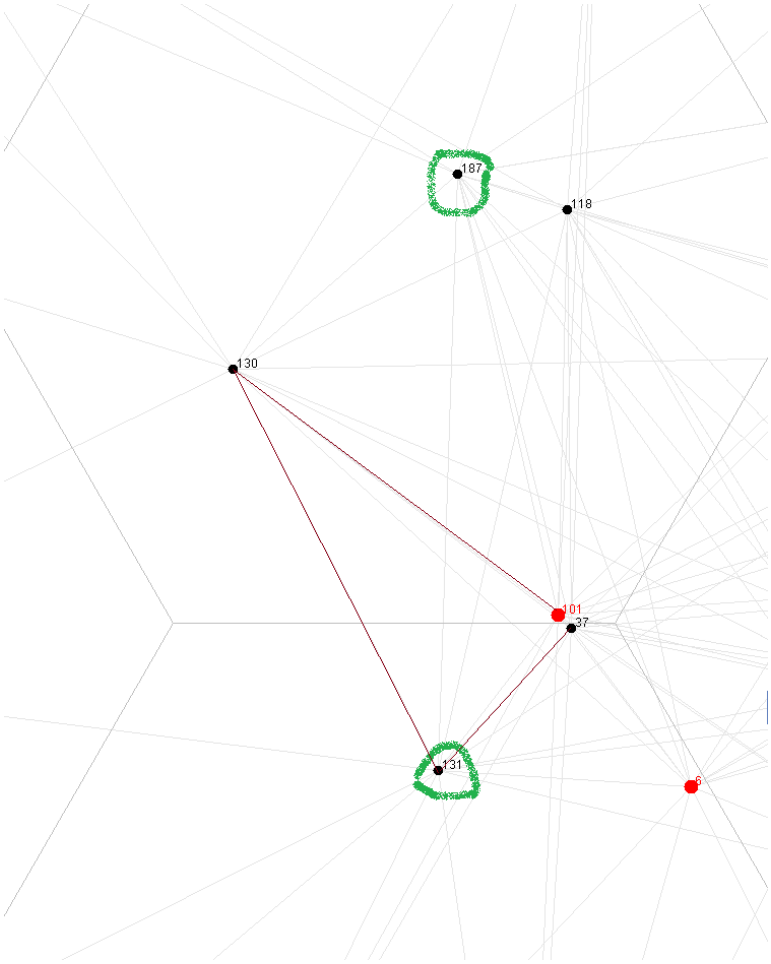


Figure 5.8: Example of a high spanning ratio

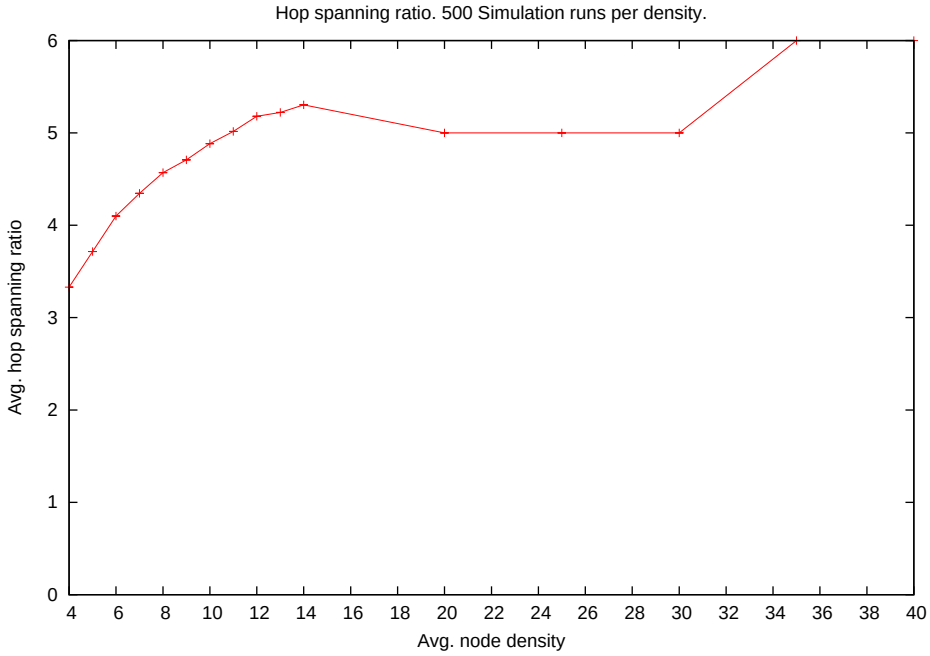


Figure 5.9: Hop spanning ratio with confidence intervals

node density. In comparison to the euclidean spanning ratio it does not increase as fast and seems to level off at six hops. The example in Figure 5.10 shows why six is the average hop spanning ratio. In the original graph the nodes 116 and 67 could reach each other with one (direct) hop. Using the overlay graph, node 116 has to send a message first to its own clusterhead (node 361) which then sends it to the border of the neighboring cluster (node 122). From there node 122 sends the message to node 512 which in turn sends it to its clusterhead (node 40). The clusterhead has a direct connection to the destination cluster (node 353). Within the destination cluster the message has to be send one final time from node 353 to the destination node 67.

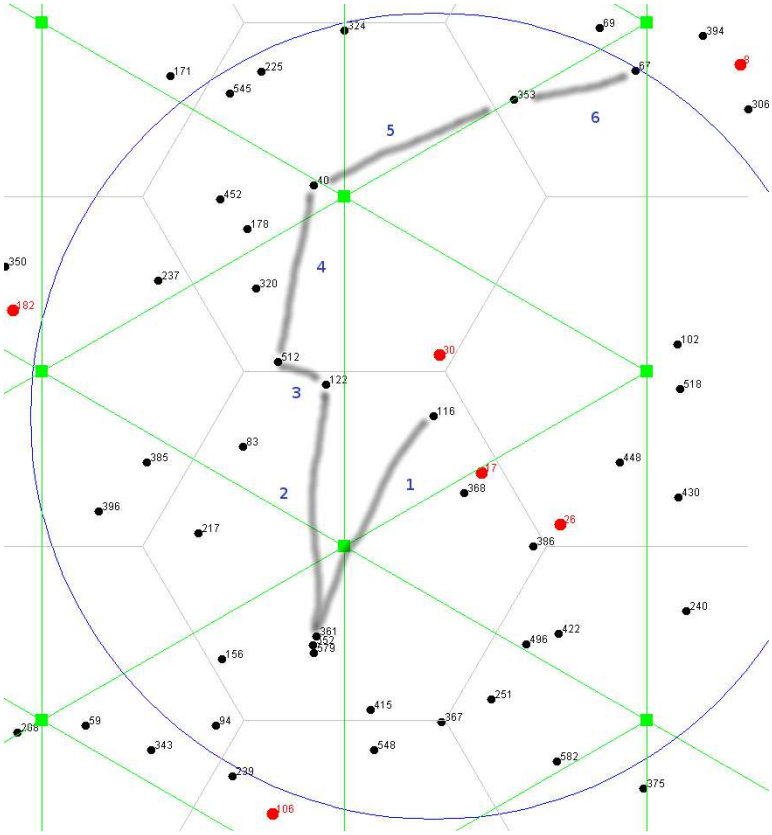


Figure 5.10: Example of a high hop spanning ratio

Chapter 6

Conclusion

The goal of this master thesis was the development of an algorithm that reactively constructs a planar overlay graph for UDGs. Since the first approach to developing such an algorithm, BLLRAP, was unsuccessful due to unprovable continued connectivity, a second algorithm, BCBP, which could be proven to work correctly, was developed. With the results of section 5.2 it can be said that the development of a reactive algorithm that reactively constructs a planar overlay graph on UDGs was successful. The BCBP algorithm is in simulations up to 70% more efficient concerning the number of messages needed and the number of nodes involved, than the original algorithm it is based upon. Furthermore the resulting overlay graph is planar (due to the nature of the algorithm), which was a key requirement. The result of this thesis contributes to the effort to beaconlessly route messages in a wireless sensor network. Although there are already many beaconless algorithms for various aspects of routing messages in wireless sensor networks, none constructs reactively a planar overlay graph on UDGs. This gap is now closed. The BCBP algorithm could for example be used in the original routing algorithm after Frey, to determine the outgoing edges of a cluster in a planar overlay graph, instead of the variant with beaconing. The actual routing algorithm would not need to be altered, but would have a more efficient basis than before. Other algorithms which use cluster based planar overlay graphs on UDGs in any capacity might profit as well from the BCBP algorithm.
The result of this master thesis offers various opportunities for improvements. First of all the suggested extension of the algorithm for QUDGs from section 4.3.7 could be proven and implemented for simulation. Since the occurrence of long edges is already minimal, overlong edges may not occur at all in reality. Furthermore the number of inactive internal cluster neighbors could be increased by developing further predictions about when a node is about to protest. Another approach could be to change the computation of the various timers used, to see what kind of combination

of timers is the most efficient for this kind of algorithm. Most efficient in the sense that the least number of messages have to be send and/or the least number of nodes have to participate. To reduce the spanning ratio, it could be computed various times with slightly shifted clusters so that nearby nodes in different clusters won't have such a huge impact on it. This approach is called Grid shifting [15].

Appendix A

Message flow diagram of BCBP

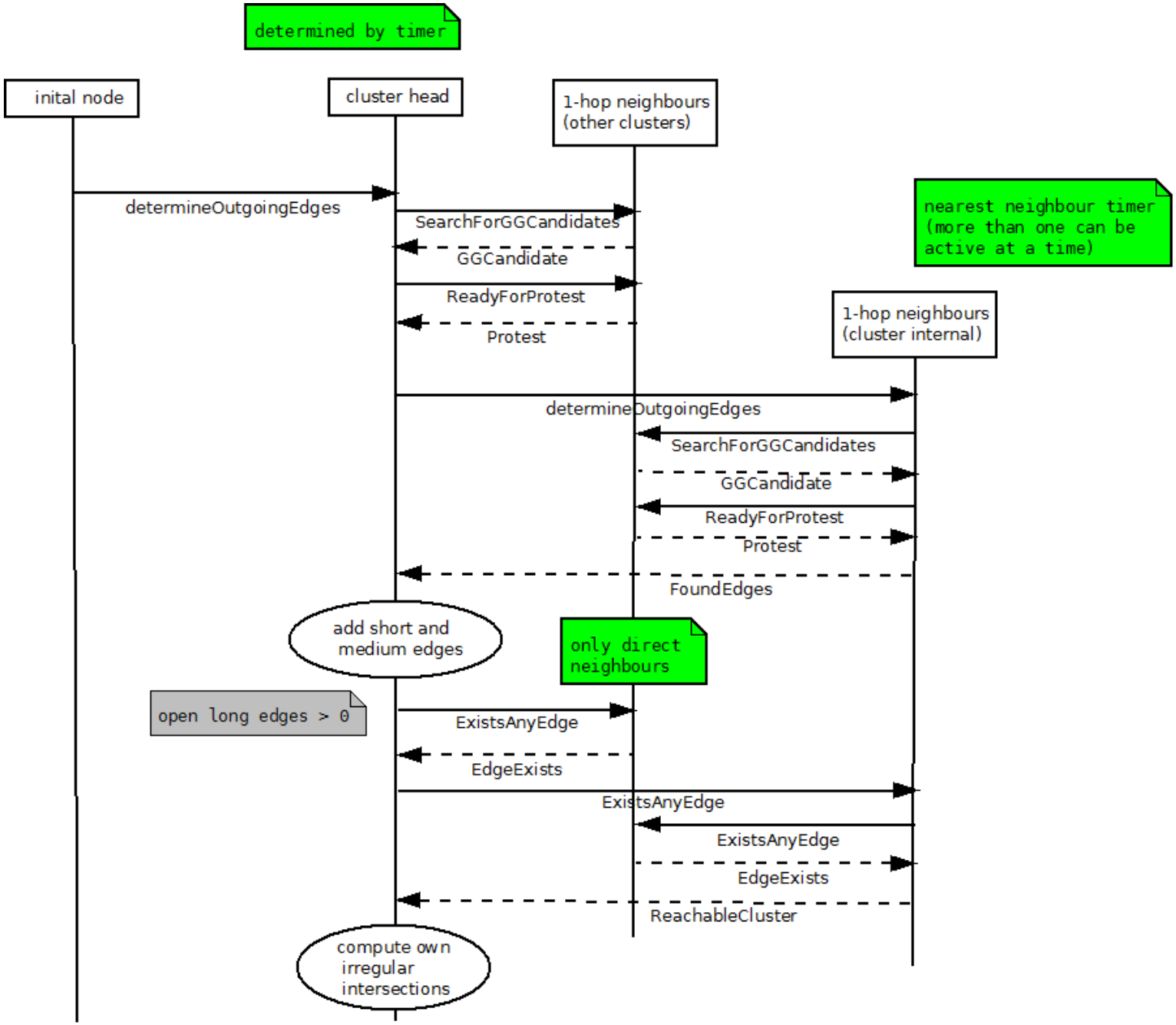


Figure A.1: Message flow of gg algorithm part 1

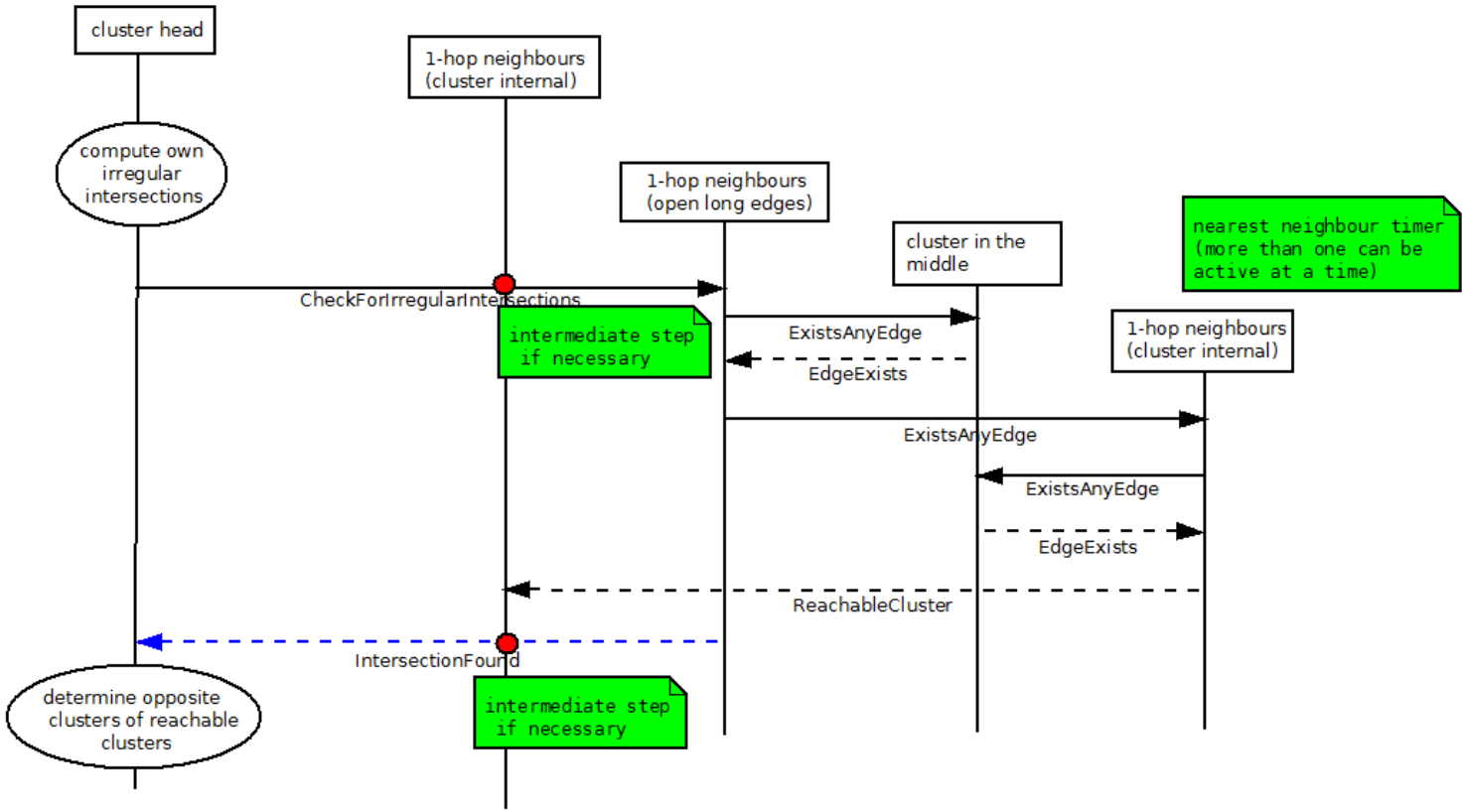


Figure A.2: Message flow of gg algorithm part 2

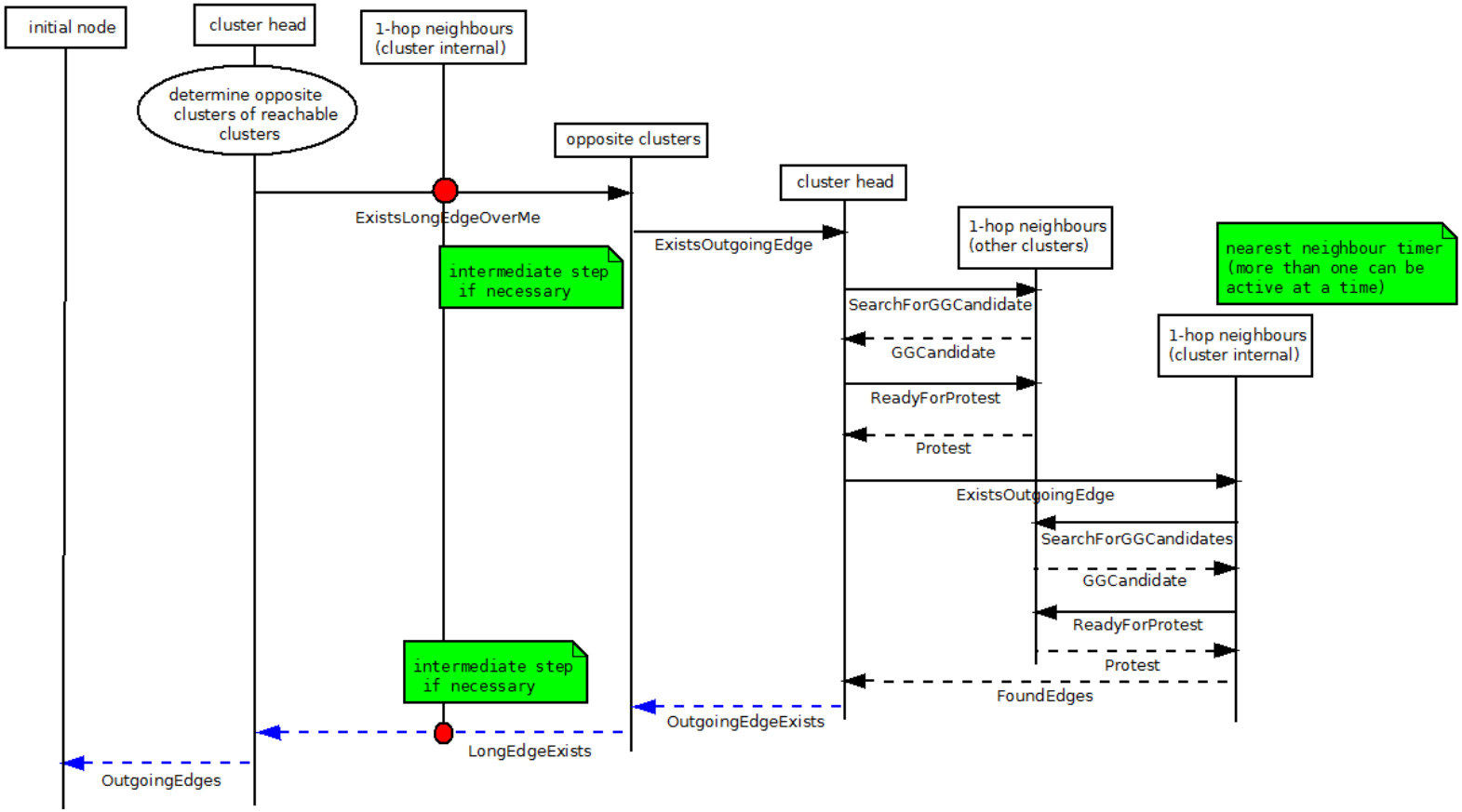


Figure A.3: Message flow of gg algorithm part 3

Bibliography

[1] Lali Barriére, Pierre Fraigniaud, Lata Narayanan, and Jaroslav Opatrny. Robust position-based routing in wireless ad hoc networks with irregular transmission ranges. *Wireless Communications and Mobile Computing*, 3(2):141–153, March 2003.

[2] Prosenjit Bose, Pat Morin, Ivan Stojmenović, and Jorge Urrutia. Routing with guaranteed delivery in ad hoc wireless networks. *Wireless networks*, 7(6):609–616, 2001.

[3] Nicolas Catusse, Victor Chepoi, and Yann Vaxès. Planar hop spanners for unit disk graphs. In *Algorithms for Sensor Systems*, pages 16–30. Springer, 2010.

[4] Mohit Chawla, Nishith Goel, Kalai Kalaichelvan, Amiya Nayak, and Ivan Stojmenovic. Beaconless position based routing with guaranteed delivery for wireless ad-hoc and sensor networks. In *Ad-Hoc Networking*, pages 61–70. Springer, 2006.

[5] Jianer Chen, A. Jiang, Iyad A. Kanj, Ge Xia, and Fenghui Zhang. Separability and topology control of quasi unit disk graphs. *Wireless Networks*, 17(1):53–67, July 2010.

[6] Hannes Frey. Geographical cluster based multihop ad hoc network routing with guaranteed delivery. In *Proceedings of the 2nd IEEE International Conference on Mobile Adhoc and Sensor Systems Conference (MASS)*, pages 510–519, Washington, D.C., US, 2005. IEEE.

[7] Hannes Frey. *Geographisches Routing: Grundlagen und Basisalgorithmen.* PhD thesis, University of Trier, 2006.

[8] Hannes Frey and Daniel Gorgen. Geographical cluster based routing in sensing-covered networks. In *Distributed Computing Systems Workshops, 2005. 25th IEEE International Conference on*, pages 885–891. IEEE, 2005.

[9] Hannes Frey and Daniel Görgen. Planar graph routing on geographical clusters. *Ad Hoc Networks*, 3(5):560–574, September 2005.

[10] Holger Füßler, Jörg Widmer, Michael Käsemann, Martin Mauve, and Hannes Hartenstein. Beaconless position-based routing for mobile ad-hoc networks. *Ad Hoc Networks*, 1:351–369, 2003.

[11] K Ruben Gabriel and Robert R Sokal. A new statistical approach to geographic variation analysis. *Systematic Biology*, 18(3):259–278, 1969.

[12] Distributed Computing Group. Sinalgo, February 2015. http://disco.ethz.ch/projects/sinalgo/index.html.

[13] Marc Heissenbüttel and Torsten Braun. A novel position-based and beacon-less routing algorithm for mobile ad-hoc networks. In *ASWN*, volume 3, pages 197–210. Citeseer, 2003.

[14] Evangelos Kranakis, Harvinder Singh, and Jorge Urrutia. Compass routing on geometric networks. In *in Proc. 11 th Canadian Conference on Computational Geometry*. Citeseer, 1999.

[15] Kevin M. Lillis, Sriram V. Pemmaraju, and Imran A. Pirwani. Topology Control and Geographic Routing in Realistic Wireless Networks. *Ad Hoc & Sensor Wireless Networks*, 6(3-4):265–297, 2008.

[16] Emi Mathews and Hannes Frey. A Localized Link Removal and Addition based Planarization Algorithm. In *Proceedings of the 13th International Conference on Distributed Computing and Networking (ICDCN)*, pages 337–350, Berlin, Heidelberg, 2012. Springer-Verlag.

[17] Julian Mosen. Ein Reaktiver Algorithmus für Geografisches Clustering. Bachelor's thesis, Universität Koblenz-Landau, 2014.

[18] David Peleg. *Distributed Computing: A Locality-sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[19] Stefan Rührup, H. Kalosha, Amiya Nayak, and Ivan Stojmenović. Message-Efficient Beaconless Georouting With Guaranteed Delivery in Wireless Sensor, Ad Hoc, and Actuator Networks. *IEEE/ACM Transactions on Networking*, 18(1):95–108, February 2010.

[20] S Son, B Blum, T He, and J Stankovic. Igf: A state-free robust communication protocol for wireless sensor networks. *Tec. Report Depart. Comput. Sci. Univ. Virginia*, 2003.

[21] Philip Sumesh. *Scalable Location Management for Geographic Routing in Mobile Ad hoc Networks by*. PhD thesis, State University of New York at Buffalo, 2005.