# Online Routing Algorithms on Geometric Graphs with Convex Substructures

by

Timothy Mott

B.Sc. Hon., Simon Fraser University, 2003.

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the Department
of
Mathematics

© Timothy Mott 2006
SIMON FRASER UNIVERSITY
Summer 2006

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

## APPROVAL

**Name:** Timothy Mott

**Degree:** Master of Science

**Title of Thesis:** Online Routing Algorithms on Geometric Graphs with Convex Substructures

**Examining Committee:** Dr. Imin Chen
Chair

______________________________

Dr. Ladislav Stacho
Senior Supervisor

______________________________

Dr. Petr Lisoněk
Supervisory Committee

______________________________

Dr. Luis Goddyn
Internal Examiner

**Date of Defense:** June 9, 2006

SIMON FRASER UNIVERSITY library

# DECLARATION OF PARTIAL COPYRIGHT LICENCE

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the right to lend this thesis, project or extended essay to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users.

The author has further granted permission to Simon Fraser University to keep or make a digital copy for use in its circulating collection, and, without changing the content, to translate the thesis/project or extended essays, if technically possible, to any medium or format for the purpose of preservation of the digital work.

The author has further agreed that permission for multiple copying of this work for scholarly purposes may be granted by either the author or the Dean of Graduate Studies.

It is understood that copying or publication of this work for financial gain shall not be allowed without the author's written permission.

Permission for public performance, or limited permission for private scholarly use, of any multimedia materials forming part of this work, may have been granted by the author. This information may be found on the separately catalogued multimedia material and in the signed Partial Copyright Licence.

The original Partial Copyright Licence attesting to these terms, and signed by this author, may be found in the original bound copy of this work, retained in the Simon Fraser University Archive.

Simon Fraser University Library
Burnaby, BC, Canada

# Abstract

In this thesis we describe five new algorithms (QUASI-PLANAR, QUASI-POLYHEDRAL, QFQ, SPIRAL, DOUBLE-CROSSING) for online route discovery on several classes of geometric graphs. We propose the classes of *quasi-planar* graphs in $\mathbb{R}^2$, which consist of underlying convex embeddings with arbitrary chords added to each face, and, analogously, *quasi-polyhedral* graphs in $\mathbb{R}^3$.

QUASI-PLANAR and QUASI-POLYHEDRAL guarantee delivery on quasi-planar and quasi-polyhedral graphs, respectively. Inspired by the well-known GFG algorithm for unit disk graphs, we create a hybrid algorithm, QFQ, that uses QUASI-PLANAR as a subroutine, and guarantees delivery on quasi-planar and unit disk graphs.

SPIRAL is a geocasting algorithm for quasi-planar graphs: it visits every vertex in a specified bounded convex region.

Finally, the DOUBLE-CROSSING algorithm finds three vertex-disjoint $s, t$-paths in a convex embedding.

# Dedication

*To those inventing their own paths*

# Acknowledgements

First, I would like to thank my supervisor, Ladislav Stacho, for his clever ideas, encouragement, and sense of humour.

I am also very grateful to my family for their support throughout my studies.

Finally, thank you, Sarah, for your love and patience.

# Contents


Approval . . . . . . . . . . ii

Abstract . . . . . . . . . . iii

Dedication . . . . . . . . . . iv

Acknowledgements . . . . . . . . . . v

Contents . . . . . . . . . . vi

List of Figures . . . . . . . . . . viii

1 Introduction . . . . . . . . . . 1

1.1 Motivation . . . . . . . . . . 1

1.2 Overview . . . . . . . . . . 3

1.3 Definitions and notation . . . . . . . . . . 4

1.4 Background . . . . . . . . . . 7

1.4.1 Classifying Routing Algorithms . . . . . . . . . . 7

1.4.2 Triangulations . . . . . . . . . . 11

1.4.3 Unit Disk Graphs . . . . . . . . . . 12

1.4.4 Planar Graphs . . . . . . . . . . 14

1.4.5 Unstable and faulty unit disk graphs . . . . . . . . . . 14

1.4.6 Quality of Service (QoS) . . . . . . . . . . 16

1.4.7 Geocasting . . . . . . . . . . 16

2 Quasi-Planar Routing . . . . . . . . . . 17

2.1 Quasi-planar routing in $\mathbb{R}^2$ . . . . . . . . . . 17

2.1.1 The QUASI-PLANAR algorithm . . . . . . . . . . 19

2.1.2 Rules for choosing the next vertex . . . . . . . . . . 24

2.1.3 Analysis . . . . . . . . . . 25

2.2 QFQ: a hybrid algorithm for unit disk graphs . . . . . . . . . . . . 27
2.3 Quasi-polyhedral routing in $\mathbb{R}^3$ . . . . . . . . . . . . . . . . . . . . . . . . . 28
2.3.1 Quasi-polyhedral graphs . . . . . . . . . . . . . . . . . . . . . . . 28
2.3.2 The QUASI-POLYHEDRAL algorithm . . . . . . . . . . . . . . 30
2.3.3 The FFINIT and FFF subroutines . . . . . . . . . . . . . . . 34
3 Quasi-planar geocasting . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
3.1 The SPIRAL geocasting algorithm . . . . . . . . . . . . . . . . . . . . . 39
3.1.1 The QP-FINDBOUNDARY subroutine . . . . . . . . . . . . 44
3.1.2 The QP-PERIMETER subroutine . . . . . . . . . . . . . . . 45
4 Disjoint Routing in Convex Embeddings . . . . . . . . . . . . . . . . . . . . 51
4.1 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
4.2 Agents and scouts . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
4.3 The DOUBLE-CROSSING algorithm on a sphere . . . . . . . . . . . 55
4.4 The DOUBLE-CROSSING algorithm in $\mathbb{R}^2$ . . . . . . . . . . . . . . . 76
4.5 Improving the algorithm . . . . . . . . . . . . . . . . . . . . . . . . . 78
Appendix . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
A Memory requirements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
Bibliography . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82

# List of Figures


1.1 $\mathbf{cw}(u, v)$ and $\mathbf{ccw}(u, v)$. . . . . 6

1.2 GREEDY chooses the neighbour of $v$ that minimises distance to $t$. . . . . 8

1.3 COMPASS chooses the neighbour of $v$ that minimises angle to $t$. . . . . 8

1.4 Both GREEDY and COMPASS fail when starting from one of the $s_i$. . . . . 8

1.5 A triangulation that defeats COMPASS. . . . . 9

1.6 The proof of Theorem 1.5 . . . . . 11

1.7 Construction of the Gabriel graph. . . . . 13

1.8 The Gabriel construction fails when the graph is not strictly a unit disk graph. . . . . 15

2.1 A quasi-planar graph; one of its underlying planar graphs is drawn with bold edges. . . . . 17

2.2 The current vertex is $v$; candidates for the next vertex are $\{b_1, \ldots, b_p, a\}$. . 20

2.3 Invalid position for $t$ when $\mathcal{R}(v, x) = a$ . . . . . 23

2.4 Invalid position for $t$ when $\mathcal{R}(v, x) = b_i$ . . . . . 23

2.5 QUASI-PLANAR, GREEDY, and COMPASS can have arbitrarily bad dilation. 26

2.6 The pairwise adjacent vertices $a, b, c$ compose a 3-cycle, but not a face. . . 29

2.7 Candidates for the next vertex include $a$ and $b$, which are feasible and forward. . . . . 30

2.8 The cone $C_{xy}$. . . . . 33

2.9 $S$ is an oriented plane through $v$, and intersecting $ab$; $u$ is a point on $S$. . . 34

2.10 (a) The edge $yz$ of the 3-cycle $ayz$ intersects the triangle $\triangle abc$. (b) The 3-cycle $va_1a_2$ dominates $vb_1b_2$. . . . . . . . . . . . . . . . . . . . . . . . . . . 35

2.11 The five faces $va_1a_2, \dots, va_5a_1$ are cap faces. . . . . . . . . . . . . . . . . . . 35

3.1 The boundary edges are drawn with thick lines. . . . . . . . . . . . . . . . 40

3.2 Proof of Lemma 3.2. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

3.3 An agent starting from $s$ navigates the perimeter of the disk using the left-hand rule. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

3.4 The current and reference vertices are $v$ and $x$, respectively. . . . . . . . . 47

3.5 The boundary face $f$ with vertices $\{u_1, \dots, u_7\}$ has eight boundary edges and will be visited several times by the agent. . . . . . . . . . . . . . . . . 48

3.6 (a) From the initial vertex $s$ with inner boundary neighbour $w$, set $t \leftarrow sw \cap \partial D$. (b) Eventually the agent completes the navigation of $D$. . . . . . 49

4.1 Two internally disjoint $s,t$-walks are determined by traversing the upper and lower halves of $st$-crossing faces. . . . . . . . . . . . . . . . . . . . . . 52

4.2 Finding two disjoint $s,t$-walks is challenging on a planar graph with non-convex faces. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53

4.3 The circle $\ell$ through $s$ and $t$ is naturally partitioned into two arcs $st$ and $\overline{st}$. Faces are omitted for clarity. . . . . . . . . . . . . . . . . . . . . . . . . 56

4.4 The basic approach of the DOUBLE-CROSSING algorithm: send two agents (DOUBLEUP and DOUBLEDOWN) along $st$-crossing faces, and a third agent (SOLITARY) along $\overline{st}$-crossing faces. . . . . . . . . . . . . . . . . . . . 57

4.5 A double-crossing configuration $(f, g_1, g_2)$. . . . . . . . . . . . . . . . . . . . 59

4.6 If agents do not adjust to DCCs, collisions are inevitable. . . . . . . . . . . 60

4.7 The correct adjustments at a double-crossing configuration $C$. . . . . . . . 61

4.8 A DCC $(f, g_1, g_2)$, approached from the "safe" direction – the DCC is not active for any of the agents. . . . . . . . . . . . . . . . . . . . . . . . . . . 63

4.9 The DCCs $C_1 = (f_1, g_1, g_2)$ and $C_2 = (g_2, f_2, f_1)$ have two faces in common. 64

4.10 The correct behaviour at DCCs $C = (f, g_1, g_2)$ and $C' = (f', g'_1, g'_2)$ entangled around $t$. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65

4.11 If the agents do not maintain an active interval, entangled DCCs will induce collisions; in this example two agents visit $x$. . . . . . . . . . . . . . 65

4.12 A single-crossing configuration $(f, g)$. . . . . . . . . . . . . . . . . . . . . . . . 66

4.13 In general (*i.e.* when not at an SCC or DCC), the solitary agent can choose to cross $\ell$. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66

4.14 The active region $R_j$. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69

4.15 DOUBLEUP cannot find a $v^{\uparrow}, t$-walk without passing through $g_1$. . . . . . . 71

4.16 DOUBLEUP cannot arrive at a vertex $w$ on $g_1 \cap f$: (a) the case where it travels along at least one edge in the $k$th phase; (b) the case where it remains in place during the $k$th phase. . . . . . . . . . . . . . . . . . . . . . 73

4.17 DOUBLEUP cannot have used $w$ before arriving at $C_k$, for then $g' \cap \ell \prec g_1 \cap \ell$, contradicting the minimality of $g_1$. . . . . . . . . . . . . . . . . . . 75

4.18 A DCC $(f, g_1, g_2)$ in $\mathbb{R}^2$. . . . . . . . . . . . . . . . . . . . . . . . . . . . 77

4.19 Another DCC $(f, g_1, f_O)$ in $\mathbb{R}^2$. . . . . . . . . . . . . . . . . . . . . . . . . . 77

4.20 In general, every agent can accelerate its face routing using information provided by the scouts. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78

4.21 DOUBLEUP has arrived at $v_1$, and its child scout $S_1$ has found the $st$-crossing face $g_1$, a potential upper face; $S_1$ is currently at $v_2$. . . . . . . . . 79

4.22 The improved initial solitary agent S! uses faces below the $st$-crossing faces and travels in the same direction as the double agents. . . . . . . . . 80

# Chapter 1

# Introduction

## 1.1 Motivation

In this thesis we consider the *online routing* problem; that is, finding routes on a graph using only local information and constant memory, and without leaving mark bits on the vertices.

Online routing is fundamental to *ad hoc networks*, which consist of self-organised mobile nodes capable of communicating with nearby nodes; each node can serve as a router for its peers. Ad hoc networks have applications in a wide array of fields, including communications networks, robotics, geographic information systems, urban planning, disaster recovery, search and rescue operations, law enforcement, and for large-scale civilian projects and events. [BMSU01, LL99, PH99].

Often, the environment does not have a predetermined topology, is large and complex, and changes over time, precluding the construction of any useful global map; hence the restriction to local information and constant memory. Because of the inherent challenges in communicating over such environments, the emphasis of many online routing algorithms is simply on guaranteeing delivery, rather than, for example, finding shortest paths.

The vertices, or nodes, in such applications are often aware of their own and their neighbours' geographic coördinates, for example by using a Global Positioning System (GPS) or triangulating from control towers. The abstract model of an ad hoc network with position-aware nodes is called a *geometric graph*; routing algorithms on such

graphs take advantage of local geometric information to reach their destinations.

Communication on an ad hoc network is carried out by transferring *packets* between nodes; The packets consist of low-level information (*e.g.*, for hardware protocols), message data, and a constant amount of parameters or geometric information necessary for a given routing algorithm (*e.g.*, the position of the destination). Upon receiving packets at an intermediate (*i.e.*, non-destination) node, the node makes a local computation based on its location, its neighbours' locations, and the stored routing information. It then forwards the packet to one of its neighbours accordingly, possibly having updated the routing information. At a higher level of abstraction, we simply consider an *agent* running a routing algorithm to be responsible for delivering messages between nodes. (We often go so far as to identify an agent with the algorithm it uses, so we might say "the algorithm $\mathcal{A}$ moves to a vertex $u$", for example, meaning "the agent using Algorithm $\mathcal{A}$ moves to $u$".) For a more detailed discussion of lower-level routing issues, see the survey by Kumar *et al.* [KRD06].

For simplicity of the theoretical model, it is usually assumed that the network remains static and connected during the delivery of the message [BMSU01]. That is, the agents are capable of navigating the network and transferring packets faster than the nodes change their position or local connections. We keep this assumption for the remainder of the thesis.

The nodes in an ad hoc network have limited memory and potentially handle large numbers of agents passing through them, so we prefer that routing algorithms do not make use of any persistent memory at the nodes. In other words, the algorithms should not be able to leave any marks behind. Furthermore, most routing algorithms do not require duplication of packets, so that there is at most one copy of any message in the network [BMSU01]. The algorithms presented in this paper also have these desirable properties, except for DOUBLE-CROSSING in Chapter 4, which requires agents to be able to create, destroy, identify, and communicate with "scouts". We postpone further discussion of this issue until Section 4.2.

To allow for a theoretic approach to the online routing problem, it is typical to impose geometric conditions on (the embedding of) the graphs. For example, a communications network may be represented as a *unit disk graph*, where two nodes are adjacent if and only if they are not farther than distance 1 from each other.

In particular, there has been significant interest in developing online routing algorithms on planar graphs because of their predictable geometry. *Face routing*, introduced in [KSU99], guarantees delivery on planar graphs by successively traversing the faces of the graph.

Obviously, we cannot expect "real-world" communication networks to be planar. However, some graphs, such as unit disk graphs, can be *planarised* to allow face routing. In other words, for some classes of graphs it is possible to (locally) compute a spanning planar subgraph. For instance, a unit disk graph has a planar subgraph, the *Gabriel* subgraph, whose edges can be determined locally.

However, in practice, the unit disk graph model is idealistic and does not account for such factors as localisation errors, obstacles blocking communication, and varying transmission capabilities [KGKS05]. Thus, it is important to develop more flexible graph models, along with algorithms designed for those models.

We propose a new class of graphs, the *quasi-planar* graphs, that may have crossing edges but contain underlying convex embeddings. We also propose an analogous class of graphs in $\mathbb{R}^3$, the *quasi-polyhedral* graphs, which similarly contain an underlying structure consisting of convex polyhedra.

## 1.2 Overview

The thesis is organised as follows. The remainder of this chapter provides a background on graph models and online routing, with emphasis on planar graphs. Chapters 2, 3, and 4 introduce five new routing algorithms – QUASI-PLANAR, QUASI-POLYHEDRAL, QFQ, SPIRAL, and DOUBLE-CROSSING – that are the result of original research.

The QUASI-PLANAR and QUASI-POLYHEDRAL algorithms in Chapter 2 are routing algorithms that guarantee delivery on quasi-planar and quasi-polyhedral graphs, respectively. QUASI-PLANAR can be considered an extension of the standard face routing algorithm for planar graphs. We also show how QUASI-PLANAR can be modified to be used as a heuristic for geometric graphs in general.

The core of Chapter 2 was presented in condensed form at PerCom 2006 [KMS06]; an expanded journal article is pending. Here we give the proofs of correctness in full detail, along with a new section on the QFQ algorithm. We approach the proofs fairly

strictly, giving detailed geometric arguments to illustrate the method of reasoning common to all the proofs. We will elide some of the lower-level details in the subsequent chapters to avoid the burden of repetitive arguments.

Chapter 3 presents a *geocasting* algorithm, SPIRAL, for quasi-planar graphs. The algorithm visits every vertex in a specified convex region of the graph. The material in this chapter is intended to be expanded into a journal article.

Finally, in Chapter 4 we consider the *disjoint routing* problem. It is straightforward to find two disjoint $s, t$-paths in a convex embedding or quasi-planar graph; we provide an algorithm, DOUBLE-CROSSING, that finds three disjoint $s, t$-walks in a convex embedding. The material from this chapter is also currently being written as a journal article.

Two geometric themes are featured throughout this thesis. First, as we have mentioned, all the algorithms in Chapters 2, 3, and 4 are designed for graphs with an underlying convex structure, and we strongly take advantage of this structure in the proofs. Conversely we give examples to show the difficulty of similar problems on graphs lacking a convex structure. In fact, we will see that even convex embeddings present surprising challenges.

Second, the proof of correctness for each of our algorithms uses a measure of progress towards the destination to guarantee termination. This is especially interesting in the DOUBLE-CROSSING algorithm, where each of three agents alternately approaches the destination $t$ from the left and from the right.

A summary of the memory requirements for each algorithm appears in Appendix A.

## 1.3 Definitions and notation

In this section we formally describe the terminology and notation to appear throughout the thesis.

A *geometric graph* $G$ is an ordered pair $G = (V, E)$ where

- $V$ is a (finite) set of *vertices*, or *nodes*, that are embedded in $\mathbb{R}^d$ for some $d \in \mathbb{Z}^+$ and
- $E$ is a set of unordered pairs of vertices, called *edges*, that we associate with the

corresponding line segments between vertices.

Furthermore, every vertex $v$ is aware of its position, its *neighbourhood* $N(v) = \{u : uv \in E\}$, and the position of each of its neighbours. We identify a geometric graph with its embedding, so we can specify vertices and edges according to their geometric positions. We may occasionally abuse the distinction between objects and their positions; thus if we say, for example, that $v = p$ for some $v \in V$ and $p \in \mathbb{R}^2$, we mean that $v$ is embedded at the point $p$. However, this abusive notation is kept to a minimum, and when used, the meaning should be clear from context.

In this thesis we are interested in those graphs embedded in $\mathbb{R}^2$ or $\mathbb{R}^3$: lower dimensions are trivial, while in higher dimensions the geometry is challenging and there is a lack of obvious practical applications.

A geometric graph in $\mathbb{R}^2$ is *planar* if there are no crossing edges; that is, if the intersections of the edges occur only at their endpoints. The edges in a planar graph partition $\mathbb{R}^2$ into a set of faces $F$, and we may write $G = (V, E, F)$. Exactly one of the faces must be infinite; we call this face the *outer face* and denote it by $f_O$. If the boundary of every face is a convex polygon, we say that $G$ is a *convex embedding*. The convex hull of a set $S$ (*i.e.*, the smallest convex set containing $S$) is denoted $conv(S)$.

If the interior of an edge $e$ crosses a line or line segment $\ell$, we say that $e$ is an *$\ell$-crossing* edge. In a planar graph, a face is $\ell$-crossing if it contains an $\ell$-crossing edge.

For vertices $u$, $v$, and $w$, we denote by $\angle uvw$ the counterclockwise angle from $u$ to $w$ about $v$. Similarly, $cone(u, v, w)$ denotes the cone with apex $v$ and supporting lines through $u$ and $w$, with interior angle $\angle uvw$. For both $\angle uvw$ and $cone(u, v, w)$ we require that $v$ does not coincide with $u$ or $w$.

Define $\mathbf{cw}(u, v)$ to be the first clockwise neighbour of $u$ starting from the direction $uv$. Note that $uv$ is not required to be an edge. Similarly, $\mathbf{ccw}(u, v)$ is the first counterclockwise neighbour of $u$ starting from the direction $uv$. See Figure 1.1. These two functions can be computed locally, as long as $uv \in E$ or the location of $v$ is known.

If $G = (V, E)$ and $H = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$, we say that $H$ is a *subgraph* of $G$ and write $H \subseteq G$. If $V' = V$ then $H$ is a *spanning* subgraph of $G$.

A *walk* in $G$ is a sequence of vertices $v_0 v_1 \dots v_k$ such that $v_i v_{i+1} \in E$ for all $i < k$. A *path* is a walk where all the vertices in the walk are distinct.

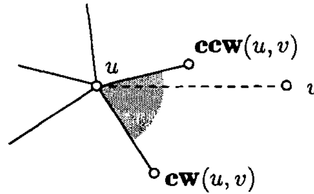


Figure 1.1: $\mathbf{cw}(u, v)$ and $\mathbf{ccw}(u, v)$.

There are two standard distance metrics on geometric graphs; namely, Euclidean distance and link (graph) distance. Unless noted otherwise, we will measure distance using the former. The *length* of a walk is the sum of the length of its edges, with repeated edges contributing accordingly. The Euclidean distance between $u, v \in V$ (*i.e.*, the length of the line segment $uv$) is denoted $dist(u, v)$, and the length of the shortest $u, v$-path is denoted $SP(u, v)$. We denote by $dist(u, S)$ the shortest distance between a vertex $u$ and a set $S$. That is, $dist(u, S) = \min_{p \in S} dist(u, p)$.

Let $G$ be a graph, $\mathcal{A}$ a routing algorithm, and $s, t \in V$. Then $\mathcal{A}(G, s, t)$ denotes the length of the walk determined by $\mathcal{A}$ when routing from $s$ to $t$; if $\mathcal{A}$ fails to reach $t$ in a finite number of steps then $\mathcal{A}(G, s, t) = \infty$.

We say that $\mathcal{A}$ *guarantees delivery* on a graph $G$ if $\mathcal{A}(G, s, t) < \infty$ for every choice of vertices $s, t \in V(G)$. On the other hand, if $\mathcal{A}$ does not guarantee delivery on $G$, then we say that $G$ *defeats* $\mathcal{A}$.

We can measure the effectiveness of an algorithm to some extent by comparing its $s, t$-walks with the shortest $s, t$-paths. $\mathcal{A}$ is *c-competitive* for a class of graphs $\mathcal{G}$ if

$$\frac{\mathcal{A}(G, s, t)}{SP(G, s, t)} \leq c$$

for every $G \in \mathcal{G}$ and all distinct pairs of vertices $s, t \in V(G)$. If there exists some constant $c$ for which $\mathcal{A}$ is $c$-competitive, then we say that $\mathcal{A}$ is *competitive*.

If there exists a path from $s$ to $t$ for all vertices $s, t \in V$, we say that $G$ is *connected*. Note that a convex embedding is necessarily connected, since the boundary of the outer face in a disconnected planar graph consists of at least two polygons. We assume for the remainder of the thesis that all graphs are connected, unless otherwise noted.

The *Voronoi diagram* [dBvKOS00] of a set of points $V \in \mathbb{R}^2$ is the partition of $\mathbb{R}^2$ into cells where each cell consists of all points closer to given point $v \in V$ than any other point in $V$. The *Delaunay triangulation* of $V$ is the (straight-line) dual of the Voronoi

diagram, *i.e.* two points in $V$ are adjacent if and only if the corresponding cells in the Voronoi diagram share an edge. In the case where some point $p \in \mathbb{R}^2$ is equidistant to some $k > 3$ points of $V$, the corresponding face in the Voronoi diagram is a $k$-gon; we can triangulate it arbitrarily.

## 1.4 Background

### 1.4.1 Classifying Routing Algorithms

Bose and Morin classify routing algorithms based on whether they require memory beyond that used for the message, and whether they are deterministic or randomised.

#### Memoryless Algorithms

A routing algorithm is *memoryless* if the choice of the next vertex from $v$ depends only on $v$, $N(v)$, and $t$. Note that for purposes of this definition, neither storing $t$ nor the message is considered to use memory, since for most applications it is obviously necessary for the agent to know when it has reached its destination and to be able to deliver the message. Also note that the neighbourhood of a vertex $v$ may be large, so in general the processor at $v$ must handle computations involving $|N(v) \cup \{v, t\}|$ vertices. In practice, say for communication networks, we may assume that this number is bounded by some constant, and that the processor at $v$ is powerful enough for such computations. The computations do not contribute to network traffic, since the only information sent to the next vertex is the message data and $t$.

The GREEDY algorithm is perhaps the simplest and most natural deterministic memoryless routing algorithm. At a vertex $v$, GREEDY chooses as its next vertex the closest neighbour of $v$ to $t$. That is,

$$\text{GREEDY}(v, N(v), t) = argmin_u\{d(u, t) : u \in N(v)\}.$$

A similar deterministic memoryless algorithm is COMPASS, which minimises angle to the destination rather than distance:

$$\text{COMPASS}(v, N(v), t) = argmin_u\{|\angle uvt| : u \in N(v)\},$$

where the angle is measured in the range $(-\pi, \pi]$.

These algorithms are illustrated in Figures 1.2 and 1.3.

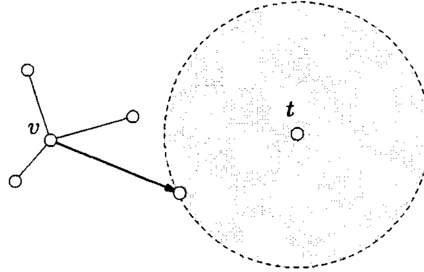


Figure 1.2: GREEDY chooses the neighbour of $v$ that minimises distance to $t$.

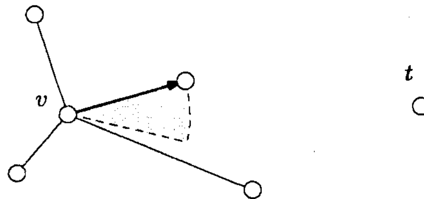


Figure 1.3: COMPASS chooses the neighbour of $v$ that minimises angle to $t$.

While GREEDY and COMPASS are useful heuristics, neither guarantees delivery on all graphs. It is straightforward to construct graphs that defeat these algorithms; naturally we may ask what conditions are necessary to guarantee delivery.

GREEDY and COMPASS do not even guarantee delivery on convex embeddings, as shown in Figure 1.4. Since $s_1$ and $s_2$ are closer to $t$ than $a_1$ and $a_2$, routing from either of the $s_i$ to $t$ will force these algorithms into an infinite loop alternating between $s_1$ and $s_2$.

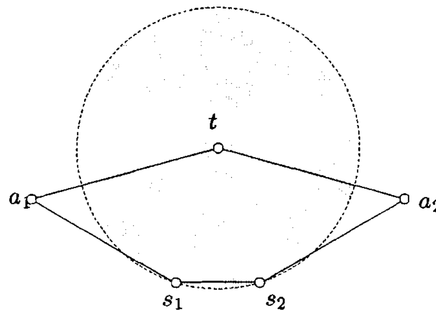


Figure 1.4: Both GREEDY and COMPASS fail when starting from one of the $s_i$.

Worse yet, GREEDY and COMPASS do not succeed on all triangulations. For example, adding the chords $a_1s_2$ and $a_1a_2$ to the graph in Figure 1.4 produces a triangulation that defeats GREEDY. The triangulation in Figure 1.5 defeats COMPASS: starting at any of the eight vertices not adjacent to $t$ results in a clockwise cycle through those vertices [BM99].

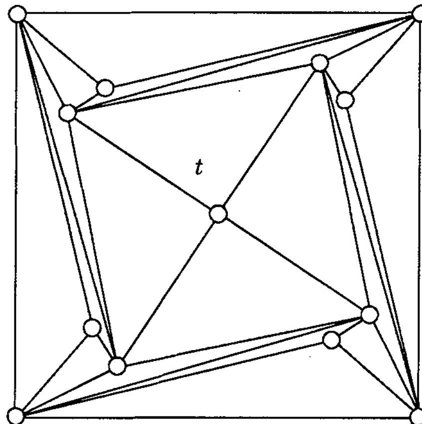


Figure 1.5: A triangulation that defeats COMPASS.

On the other hand, there are important classes of triangulations for which these algorithms do guarantee delivery.

**Theorem 1.1** *[BM99]* GREEDY *guarantees delivery on every Delaunay triangulation.* □

A *regular triangulation* is a triangulation obtained by orthogonal projection of the faces of the lower hull of a 3-dimensional polytope onto the plane.

**Theorem 1.2** *[BM99]* COMPASS *guarantees delivery on every regular triangulation.* □

Bose *et al.* propose two variants of COMPASS that succeed on larger classes of graphs: GREEDY-COMPASS, which chooses the closest to $t$ of $\mathbf{cw}(v,t)$ and $\mathbf{ccw}(v,t)$, and RANDOM-COMPASS, which randomly chooses between $\mathbf{cw}(v,t)$ and $\mathbf{ccw}(v,t)$.

**Theorem 1.3** *[BMB⁺00]* GREEDY-COMPASS *guarantees delivery on every triangulation.* □

**Theorem 1.4** *[BMB⁺00]* RANDOM-COMPASS *guarantees delivery on every convex subdivision.* □

Incidentally, note that a random walk guarantees delivery on every connected graph, so the latter result should not be a surprise. Naturally we are more interested in algorithms that make steady, rather than random, progress towards the destination. We observed earlier that neither GREEDY nor COMPASS guarantees delivery on all triangulations, suggesting that deterministic memoryless algorithms are not likely to be robust. Bose *et al.* affirm this hypothesis; their argument is as follows.

**Theorem 1.5** *[BMB⁺00] Every deterministic memoryless online routing algorithm is defeated by some convex embedding.*

*Proof.* Consider the three graphs $G_1, G_2, G_3$ shown in Figure 1.6. Common to all three are the vertices $v_0, \ldots, v_{15}$ arranged in order on a regular 16-gon centred at the origin, the edges of this 16-gon, and a destination vertex $t$ at the origin. Note that the even-numbered vertices $v_0, v_2, \ldots, v_{14}$ all have degree 2.

Towards a contradiction, suppose there exists a deterministic memoryless routing algorithm $\mathcal{A}$ that succeeds on any convex subdivision. Since $v_0, v_2, \ldots, v_{14}$ have the same neighbours in all three graphs, $\mathcal{A}$ makes the same decision at any particular even-numbered vertex. We can therefore consider the behaviour of $\mathcal{A}$ at these vertices regardless of the graph on which it is routing.

Colour each $v_{2i}$ black or white according to whether $\mathcal{A}$ moves counterclockwise or clockwise from $v_{2i}$ around the 16-gon, respectively. We claim that all even-numbered vertices must have the same colour. If not, there exist two vertices $v_{2i}$ and $v_{2i+2}$ (subscripts are considered mod 16) such that $v_{2i}$ is black and $v_{2i+2}$ is white. Then, taking $s = v_{2i}$ in $G_1$, the algorithm gets stuck either between $v_{2i}$ and $v_{2i+1}$, or $v_{2i+1}$ and $v_{2i+2}$, a contradiction.

Therefore we can assume that all even-numbered vertices are black. Now consider the graph $G_2$. If $s = v_1$, then $\mathcal{A}$ cannot visit $x$ after $v_1$, or it would cycle among the vertices $\{v_{12}, v_{13}, v_{14}, v_{15}, v_0, v_1, x\}$ without reaching $t$. Rotating the edges in $G_2$, we can make similar arguments for all odd-numbered vertices.

However, this implies that in the graph $G_3$, if we take $s$ to be any of the $v_i$, then

$\mathcal{A}$ never enters the interior of the graph. Therefore no deterministic memoryless algorithm guarantees delivery on all convex subdivisions. (Note that the figure can be slightly modified to make the same argument for strictly convex subdivisions.) ■

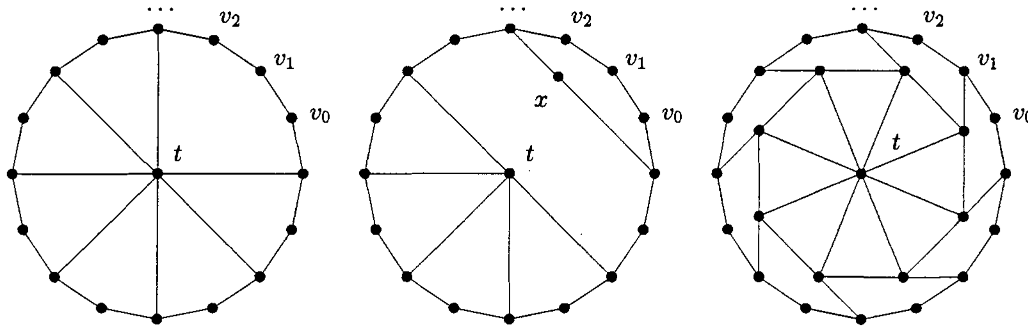


Figure 1.6: The proof of Theorem 1.5

**Constant Memory Algorithms**

A routing algorithm runs with *constant memory* on a graph of order $n$ if the choice of the next vertex from $v$ depends only on $v$, $N(v)$, $t$, and $O(\log n)$ bits of of memory. Typically the memory is used to store vertex labels; thus, equivalently, we can say that a constant memory algorithm uses $O(1)$ memory, where it is understood that one memory slot takes $\log n$ bits.

### 1.4.2 Triangulations

Given a set of points $V$, a *minimum-weight* triangulation on $V$ is a triangulation minimising the sum of the edge lengths. A *greedy* triangulation on $V$ is constructed by starting with an empty edge set and repeatedly adding the shortest edge that does not cross any other previously-added edge. Bose *et al.* prove the following fundamental results [BMSU01]:

- under the Euclidean metric, no deterministic routing algorithm is $o(\sqrt{n})$-competitive for all triangulations;
- under the link metric, no deterministic routing algorithm is $o(\sqrt{n})$-competitive for all Delaunay, greedy, or minimum-weight triangulations.

An *ear* in a triangulation is a vertex of degree 2. Bose *et al.* [BM01] present a 9-competitive algorithm for triangulations with two ears, and use this as a basis for the following result on an important class of triangulations.

Let $0 < \alpha \leq \pi/2$. For an edge $e$ of a triangulation $T = (V, E)$, consider the two isosceles triangles $t_1$ and $t_2$ with base $e$ and base angle $\alpha$. Then $e$ satisfies the *diamond property* with parameter $\alpha$ if one of $t_1$ or $t_2$ does not contain any vertex $v \in V$ in its interior. If this property holds for every $e \in E$, we say that $T$ satisfies the diamond property.

The set of such graphs satisfying the diamond property includes several important classes of triangulations, such as Delaunay triangulations, minimum weight triangulations, and greedy triangulations [DJ89].

**Lemma 1.6** *[BM01] Given a triangulation $T = (V, E)$ satisfying the diamond property with parameter $\alpha$, there exists a constant $d_\alpha$ (depending on $\alpha$) such that for all $x, y \in V$, $SP_T(x, y)/dist_T(x, y) \leq d_\alpha$.* □

**Theorem 1.7** *[BM01] There is a $9d_\alpha$-competitive O(1)-memory online routing algorithm that guarantees delivery on triangulations satisfying the diamond property.* □

### 1.4.3 Unit Disk Graphs

A *unit disk graph* is a geometric graph in which the neighbourhood of a vertex $v$ consists of all those vertices within a circle of radius $r$ centred at $v$. That is, $U$ is a unit disk graph if the edge set is $E(U) = \{uv : d(u, v) \leq r\}$.

The unit disk graph is a natural model for ad hoc wireless networks under idealised settings: assume that all nodes are embedded in a plane, and that every node is equipped with a communication device capable of sending and receiving messages within a constant broadcast radius $r$. Then the graph representing the communication network is precisely the unit disk graph.

#### The Gabriel Graph

Routing on a unit disk graph $U$ is typically achieved by restricting the agent to the *Gabriel graph* $G$ of $U$, which is a subgraph of $U$ with several important properties.

The Gabriel graph [GS69] of $U$ is defined as follows. Let $disk(u, v)$ be the disk with diameter $uv$. Then $G$ is the subgraph of $U$ such that $uv \in E(G)$ if and only if $disk(u, v)$ contains no other vertices of $U$. See Figure 1.7.

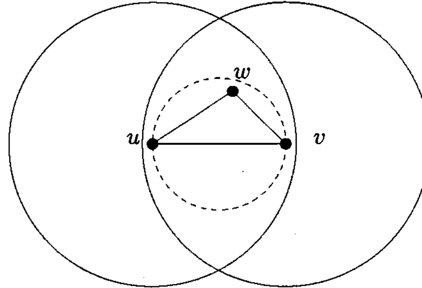


Figure 1.7: Construction of the Gabriel graph. Both $u$ and $v$ detect that $w \in disk(u, v)$; hence $uv$ is removed. The large circles around $u$ and $v$ are unit disks representing communication range.

It is possible to construct the Gabriel graph with a distributed algorithm using only local information: each node $u$ asks each of its neighbours $v$ whether it has any common neighbours with $u$ within $disk(u, v)$; if so, $u$ deletes the edge $uv$.

An agent can therefore route on the Gabriel graph by ignoring any edges not in $G$.

**Algorithm 1** Gabriel

```
1: procedure GABRIEL(U, s, t)                    ▷ U is a unit disk graph
2:     for each u ∈ N(v) do
3:         if disk(u, v) ∩ (N(v) \ {u, v}) ≠ ∅ then
4:             delete (u, v)
5:         end if
6:     end for
7: end procedure
```

The GABRIEL algorithm as given here requires $O(d^2)$ time at vertex $v$, where $d$ is the degree of $v$. This can be reduced to $O(d \log d)$ by constructing the Voronoi diagram and Delaunay triangulation of $N(v) \cup \{v\}$ and keeping only the edges of the Delaunay triangulation that intersect the corresponding edges of the Voronoi diagram.

**Theorem 1.8** *If $U$ is a connected unit disk graph then* GABRIEL *computes a connected, spanning planar subgraph of $U$. The cost of the computation at vertex $v \in V(U)$ is $O(d \log d)$ where $d$ is the degree of $v$.* □

It follows from this result that for routing on unit disk graphs, it is sufficient to develop algorithms that guarantee delivery on planar graphs.

### 1.4.4 Planar Graphs

A planar graph $G$ natural subdivides the plane into *faces*. Every edge $uv$ belongs to two faces, which can be found locally by iterating **cw** (resp. **ccw**) until returning to $uv$. This motivates the concept of *face routing*, introduced in [KSU99]: traverse the current face, noting which edges cross the line $st$. Then travel to the edge with the closest crossing to $t$, and repeat on the next face. We refer to this method as FACE-1.

If all the faces are convex, the edge with the closest crossing to $t$ will be the first edge crossing $st$. Therefore the algorithm can progress to the next face immediately upon finding a crossing edge. We call this method FACE-2.

The GREEDY-FACE-GREEDY (GFG) algorithm combines GREEDY with FACE-1 for routing on unit disk graphs. GFG uses greedy routing as long as possible; if it reaches a vertex $v'$ whose neighbours are all farther from $t$ than $v$ is, GFG switches into face-routing mode on the Gabriel subgraph. It traverses the current face until it reaches a vertex $v''$ such that $d(v'', t) < d(v', t)$, at which point it reverts to greedy mode.

### 1.4.5 Unstable and faulty unit disk graphs

Unit disks provide a powerful and convenient model for communication networks. Ideally, a wireless network should be representable as a unit disk graph; in practice, however, there will be a number of reasons to expect discrepancies.

First, it is unrealistic to expect that every node has the same power. Some nodes may be stationary bases with a very large communications range, while mobile nodes using battery power have a smaller range, especially when limiting themselves to conserve power.

Differences in transmission range can introduce one-way edges in the communications network: a node may be powerful enough to receive a message from a weak neighbour, but the neighbour too weak to receive messages itself.

The Gabriel graph, which is normally very useful for routing on unit disk graphs, is not robust with respect to these issues. Figure 1.8 shows a situation where $u$ and $v$

disagree on whether the edge $uv$ should be kept or discarded [BFN01]: $u$ claims that it should be removed, while $v$ claims that it should be kept.

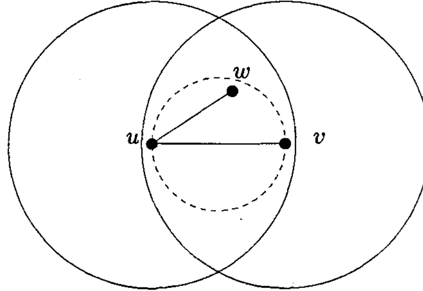


Figure 1.8: The Gabriel construction fails when the graph is not strictly a unit disk graph. Here only $u$ detects that $w \in disk(u, v)$; hence $u$ and $v$ disagree on the status of $uv$.

On a network with variable transmission range, the local algorithm for constructing the Gabriel graph may produce either a non-planar graph or a disconnected graph, both of which are undesirable.

Barrière *et al.* address these issues in [BFN01] as follows. Let $r, R \in \mathbb{R}$ be minimum and maximum communication ranges, respectively. The network is represented by an undirected geometric graph $G$ where $uv \in E(G)$ if $d(u, v) \leq r$, $uv \notin E(G)$ if $d(u, v) > R$, and $uv$ may or may not be in $E(G)$ if $r < d(u, v) \leq R$. Then a routing protocol is introduced that guarantees delivery if the ratio $R/r$ between communication ranges is at most $\sqrt{2}$.

We can also model poor communication conditions with *faultiness*: suppose that each edge in the graph has the potential to malfunction for some reason. For instance, in addition to the issues mentioned above, it may be impossible for two nearby nodes to communicate with each other if there are obstacles (mountains or tall buildings, for example) blocking the way; bad weather or other interference may also intermittently prevent communication between certain nodes.

Formally, let $U$ be a unit disk graph, and $\epsilon \in [0, 1)$ some constant. Let $F$ be a graph obtained by independently removing each edge from $U$ uniformly at random with probability $\epsilon$. Then we say that $F$ is a *faulty* unit disk graph with faultiness $\epsilon$. Of course, faultiness could be defined for any graph, but we are particularly interested in faulty unit disk graphs since they are extremely easy to generate, and provide a simple testing ground for GFG, for example. We discuss this further in Section 2.2.

### 1.4.6 Quality of Service (QoS)

We may wish not only to find a route between two specified vertices, but to find such a route satisfying certain constraints or optimising several parameters; these metrics are collectively known as *quality of service* (QoS) metrics. For instance, in communication networks it is desirable to find paths with small delay while keeping the bandwidth as low as possible at every vertex. For a more comprehensive overview, see, for example, the survey by Paul and Raghavan [PR02].

In Section 2.1.2 we provide a simple example of QoS routing by using a parameter $load(v)$ on each vertex $v$ to choose the next vertex. We consider $load(v)$ to be an abstraction of the total bandwidth through $v$ over a period of time, and assume that the load information of a vertex $v$ and its neighbours is available to an agent at $v$.

### 1.4.7 Geocasting

The *geocasting* problem generalises the routing problem so that the destination is a region (*i.e.*, every node within a region) rather than a single node. One solution to the geocasting problem is to use a *traversal* algorithm to enumerate every node. Morin, in his PhD thesis [Mor01], describes such an algorithm, FACE-GEOCAST, for planar graphs.

**Theorem 1.9** *[Mor01]* FACE-GEOCAST *is an* $O(1)$ *memory geocasting algorithm with delivery time at most* $40m(H_m - 1)$*, where* $m$ *is the total number of edges of the faces intersecting the destination region* $R$*, and* $H_k = \sum_{i=1}^{k} 1/i$ *is the* $k$*th harmonic number.*
□

The SPIRAL geocasting algorithm that we describe in Chapter 3 is designed for quasi-planar graphs. It would be interesting to compare running times between SPIRAL and FACE-GEOCAST on the class of graphs for which both algorithms guarantee delivery, *i.e.*, on convex embeddings.

# Chapter 2

# Quasi-Planar Routing

## 2.1 Quasi-planar routing in $\mathbb{R}^2$

Let $G = (V, E, F)$ be a planar graph with vertex set $V$, edge set $E$, and face set $F$. A *convex embedding* of $G$ is a straight-line embedding into the plane such that the boundary of every face is a convex polygon; we will associate $G$ with its convex embedding. For the remainder of the paper we assume that such a graph $G$ has no three collinear vertices.

Let $G = (V, E, F)$ be a convex embedding, and construct a new graph $Q$ by adding chords to the faces of $G$ except for the outer face $f_O$. That is $Q = (V, E \cup E')$, where each edge $e \in E'$ joins two vertices of some face $f \in F \setminus \{f_O\}$. We call such a graph $Q$ a *quasi-planar graph*: there may be many crossing edges, but a facial structure remains. Figure 2.1 illustrates an example of a quasi-planar graph.

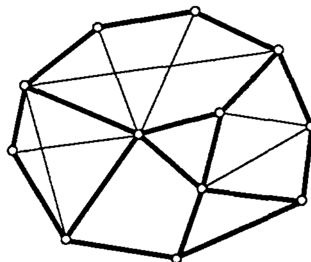

Figure 2.1: A quasi-planar graph; one of its underlying planar graphs is drawn with bold edges.

We refer to $G$ as an *underlying planar graph* of $Q$, and say that the faces $f_i \in F$ of $G$ are *underlying faces* of $Q$. Note that an underlying planar graph is not necessarily unique for a given quasi-planar graph. In general, there will be many possibilities: consider, for example, a triangulation of a convex $n$-gon. Each of the $n-3$ interior edges may or may not be a chord, so there are $2^{n-3}$ possible ways to interpret the $n$-gon with respect to the underlying planar graph.

For the purposes of our routing algorithm, however, it suffices to know that such a graph $G$ exists; the particular choice of $G$ is irrelevant and will not affect the behaviour of the algorithm. The existence of the graph $G$ is used only in proofs of correctness of the algorithm.

Recall that $\mathbf{cw}(u, v)$ is the first clockwise neighbour of $u$ starting from the direction $uv$. The edges $uv_1$ and $uv_2$ are *radially adjacent* if $v_2 = \mathbf{cw}(u, v_1)$ or $v_2 = \mathbf{ccw}(u, v_1)$. Observe that if $uv_1, uv_2 \in E$ are radially adjacent edges then some underlying face $f$ contains $u$, $v_1$, and $v_2$. Depending on the choice of the underlying planar graph $G$, the edges $uv_i$ may be outer edges or chords of $f$, but again, this distinction is not important.

Let $u, v, w_1, w_2, \ldots, w_p \in V$. Then $w_1, w_2, \ldots, w_p$ form a *clockwise sequence* around $u$ from $v$ if they are the first $p$ consecutive clockwise neighbours of $u$ starting from the direction determined by $v$. Note that $v$ is not necessarily adjacent to $u$. A *counterclockwise sequence* is defined analogously.

We denote by $uv$ the line segment through vertices $u$ and $v$; it will be clear from context whether $uv$ refers to an edge or a line segment. The line segment $st$ separates the vertex set into two subsets $V_A$ and $V_B$ that we can think of as containing vertices "above" and "below" $st$, respectively. Specifically, $V_A = \{v \in V : 0 < \angle tsv < \pi\}$ and $V_B = \{v \in V : \pi < \angle tsv < 2\pi\}$, and $V = \{s, t\} \cup V_A \cup V_B$.[1] Since $G$ is represented by a convex embedding and using the assumption that $st \notin E$, it follows that both $V_A$ and $V_B$ are non-empty. If a vertex $v$ knows the geometric locations of $s$ and $t$, it is a fast local computation to determine whether $v \in V_A$ or $v \in V_B$.

**Lemma 2.1** *Let $Q$ be a quasi-planar graph with $s, t \in V$ given, and let $v \in V_A$. If $N(v) \cap V_A = \emptyset$ then $vs, vt \in E$. Similarly, for a vertex $v \in V_B$, if $N(v) \cap V_B = \emptyset$ then $vs, vt \in E$.*

[1]The definitions of $V_A$ and $V_B$ depend on the choice of $s, t$; however, their reference will be omitted as it can be easily understood from the context.

*Proof.* We argue by contradiction: suppose there exists a vertex $v \in V_A$ such that $N(v) \cap V_A = \emptyset$, and $vs \notin E$. Index the neighbours $u_1, u_2, \ldots, u_p$ of $v$ such that $\angle u_1 v u_2 < \angle u_1 v u_3 < \ldots < \angle u_1 v u_p$. By convexity of the outer underlying face, it follows that no vertex lies outside $cone(u_1, v, u_p)$. Therefore, $s$ is contained within the convex hull of $\{v, u_i, u_{i+1}\}$ for some $i$. But $v$, $u_i$, and $u_{i+1}$ are all on the same underlying face, which, being convex, must have an empty interior. This shows that $v$ must be adjacent to $s$; similarly, $vt \in E$.

The same argument applies to a vertex in $V_B$. ■

### 2.1.1 The QUASI-PLANAR algorithm

We now describe an $O(1)$-memory routing algorithm that guarantees delivery on quasi-planar graphs. For the remainder of this section we will omit reference to the choice of underlying planar graph for a given quasi-planar graph; the results hold for any such choice. The QUASI-PLANAR algorithm traverses vertices within the underlying $st$-crossing faces, using the left- and right-hand rules (*i.e.*, using the functions **ccw** and **cw**) when $v \in V_A$ and $v \in V_B$, respectively; see Algorithm 2.

Routing from $s$ to $t$ is trivial when $s = t$ or $st \in E$; we therefore assume that $s$ and $t$ are distinct and non-adjacent, and for brevity in the following algorithm we refrain from explicitly checking for the trivial cases.

As is typical of other algorithms using the face routing technique, the QUASI-PLANAR algorithm only requires enough memory to remember $s$, $t$, and one other reference vertex $x$; this latter vertex is used to store information about the current underlying face. Whenever the current vertex $v$ is in $V_A$, $x$ will be in $V_B$, and *vice versa*.

Finally, QUASI-PLANAR requires a rule $\mathcal{R}$ that will determine the next vertex from the neighbours of the current vertex $v$. First suppose $v \in V_A$, and hence $x \in V_B$. Let $b_1, b_2, \ldots, b_p, a$ be a counterclockwise sequence around $v$ from $x$, where $p \geq 0$, $b_i \in V_B$, and $a \in V_A$. Although the set $\{b_1, b_2, \ldots, b_p\}$ may be empty (that is, $p = 0$ is possible), Lemma 2.1 guarantees the existence of $a$. We require that the function $\mathcal{R}(v, x)$ evaluate to an element from the (non-empty) set $\{b_1, b_2, \ldots, b_p, a\}$; see Figure 2.2.

For sake of simplicity, we abuse notation and also refer to $\mathcal{R}(v, x)$ when $v \in V_B$ and $x \in V_A$, with the understanding that $\mathcal{R}$ is symmetric about $st$. That is, $\mathcal{R}(v, x) \in \{a_1, a_2, \ldots, a_q, b\}$ where $a_1, a_2, \ldots, a_q, b$ is a clockwise sequence around $v$ from $x$, $q \geq 0$,

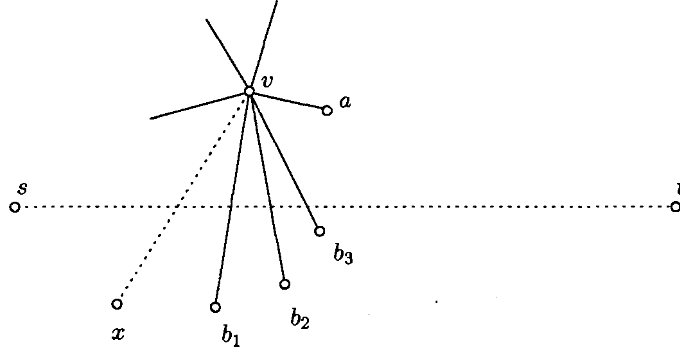


Figure 2.2: The current vertex is $v$; candidates for the next vertex are $\{b_1, \ldots, b_p, a\}$.

$a_i \in V_A$, and $b \in V_B$.

As we will prove shortly, the particular choice of $\mathcal{R}$ does not affect the correctness of the algorithm on quasi-planar graphs.

**Theorem 2.2** *Given a quasi-planar graph $Q$ and distinct, non-adjacent vertices $s, t \in V(Q)$, the* QUASI-PLANAR *algorithm successfully routes from $s$ to $t$.*

*Proof.* We will show that $v$ and $x$ are on the same underlying face during the execution of QUASI-PLANAR. Furthermore, let $l_k$ denote the point of intersection of $vx$ with $st$ after the $k$th iteration of the `while` loop. We will also show that if $v \neq t$ after the $k$-th iteration, then $l_k$ exists and $s \prec l_0 \prec \ldots \prec l_k \prec t$ where $\prec$ is the natural ordering along $st$.

The intersection points $l_k$ are determined by pairs of distinct vertices in $Q$, so the sequence $l_0, l_1, \ldots$ has at most $\binom{|V|}{2}$ terms. The `while` loop iterates as long as $vt \notin E$, resulting in a new intersection point with each iteration. Therefore, since this sequence of points is finite, it follows that after some iteration, $vt \in E$. The `while` loop then terminates and $v$ reaches $t$ at step 25.

Thus, it remains to prove the above two claims (Claims 2.1 and 2.4 in what follows). We proceed by induction on $k$, the number of iterations of the `while` loop in steps 4–24.

**Claim 2.1** *Vertices $v$ and $x$ are on the same underlying face.*

*Proof.* This is certainly true after steps 2 and 3. For $k \geq 1$, first suppose $v \in V_A$. If $\mathcal{R}(v, x) = a$, then the argument is as follows. The vertex $a$ is the first neighbour of $v$

**Algorithm 2** Quasi-Planar Routing

1: **procedure** QUASI-PLANAR($Q, s, t, \mathcal{R}$)
2: $v \leftarrow$ **ccw**$(s,t)$
3: $x \leftarrow$ **cw**$(s,t)$
4: **while** $vt \notin E$ **do**
5: **if** $v \in V_A$ **then**
6: Find the counterclockwise sequence $b_1, b_2, \ldots, b_p, a$ around $v$ from $x$, where $p \geq 0$, $a \in V_A$ and $b_i \in V_B$, $1 \leq i \leq p$.
7: **if** $\mathcal{R}(v,x) = a$ **then**
8: $x \leftarrow b_p$
9: $v \leftarrow a$
10: **else** ▷ in this case $\mathcal{R}(v,x) = b_k$ for some $k$, $1 \leq k \leq p$
11: $x \leftarrow v$
12: $v \leftarrow b_k$
13: **end if**
14: **else** ▷ $v \in V_B$
15: Find the clockwise sequence $a_1, a_2, \ldots, a_q, b$ around $v$ from $x$, where $q \geq 0$, $b \in V_B$ and $a_i \in V_A$, $1 \leq i \leq q$.
16: **if** $\mathcal{R}(v,x) = b$ **then**
17: $x \leftarrow a_q$
18: $v \leftarrow b$
19: **else** ▷ in this case $\mathcal{R}(v,x) = a_k$ for some $k$, $1 \leq k \leq q$
20: $x \leftarrow v$
21: $v \leftarrow a_k$
22: **end if**
23: **end if**
24: **end while**
25: $v \leftarrow t$
26: **end procedure**

counterclockwise from $vb_p$, so after the updates $x \leftarrow b_p$ and $v \leftarrow a$, the vertices $v$ and $x$ will be on the same underlying face. If $\mathcal{R}(v,x) = b_k$, $1 \leq k \leq p$, then after the update, $v$ and $x$ will be adjacent and hence must be on the same underlying face.

If $v \in V_B$ the argument is similar. □

**Claim 2.2** *If $v \in V_A$, then for every $0 \leq i < p$, the vertices $v, b_i, b_{i+1}$ are on the same underlying face. Moreover, the vertices $v, b_p, a$ are on the same underlying face. Similarly if $v \in V_B$, then for every $0 \leq i < q$, the vertices $v, a_i, a_{i+1}$ are on the same underlying face, and the vertices $v, a_q, b$ are on the same underlying face.*

*Proof.* We only consider the case $v \in V_A$ in detail; the other case is similar. For $i \geq 1$ the

statement follows since $b_{i+1}$ is the first neighbour of $v$ counterclockwise from $vb_i$. Thus, suppose $i = 0$, so we must show that $v$, $x$, and $b_1$ are on the same underlying face. If $vx \in E$, the argument is the same as above: $vx$ and $vb_1$ are radially adjacent edges. On the other hand, if $vx \notin E$, let $u = \mathbf{cw}(v, b_1)$. Then the vertices $v, b_1, u$ lie on the same underlying face $f$. Now, since $x$ is contained in $cone(u, v, b_1)$, and from Claim 2.1, it follows that $x$ also lies on $f$.

The same reasoning shows that $v$, $b_p$, and $a$ are on the same underlying face. ☐

**Claim 2.3** *If $v \in V_A$, then $\angle svx < \angle svt$, and similarly if $v \in V_B$, then $\angle xvs < \angle tvs$. That is, the line segments $vx$ and $st$ intersect.*

*Proof.* First, when $k = 0$, note that from the assumptions that $st \notin E$ and no three vertices are collinear, it follows from the convexity of the underlying faces that $v \in V_A$ and $x \in V_B$ exist and are well-defined after the initialisation (steps 2–3). By choice of $v$ and $x$, it is clear that $v = \mathbf{ccw}(s, x)$, so $s$, $v$, and $x$ all lie on a common underlying face $f$. If $\angle svx > \angle svt$, there are two possibilities: either $\pi < \angle xsv < 2\pi$, or $t$ is in the convex hull of $s$, $v$, and $x$. Because $f$ is convex and the angle $\angle xsv$ is an interior angle, $0 < \angle xsv < \pi$, eliminating the first case. On the other hand, $t$ cannot be in the interior of $f$, so $t$ is not in the convex hull of $s$, $v$, and $x$. Therefore $\angle svx < \angle svt$, establishing the basis of the induction.

Now assume that after $k$ iterations of the `while` loop, the desired property holds. By symmetry, we may without loss of generality assume that currently $v \in V_A$, and consequently $x \in V_B$.

During the $k + 1$-st iteration, first suppose that $\mathcal{R}(v, x) = a$. Then $v$ and $x$ will be assigned $a$ and $b_p$ respectively, so we must show that $\angle sab_p < \angle sat$. Towards a contradiction, suppose that $\angle sat < \angle sab_p$. Then $t$ lies within the convex hull of $v$, $b_i$, and $b_{i+1}$ for some $0 \leq i < p$, or within the convex hull of $v$, $b_p$, and $a$; see Figure 2.3. But each of these triples lies on an underlying face, by Claim 2.2, which by convexity cannot contain $t$, a contradiction.

If, on the other hand, $\mathcal{R}(v, x) = b_i$ for some $i > 0$, then $v$ and $x$ will be assigned $b_i$ and $v$, respectively, and we must show that $\angle tb_iv < \angle tb_is$. To this end, suppose that $\angle tb_is < \angle tb_iv$. Then either $\pi < \angle xvt < 2\pi$ or $0 < \angle xvt < xvb_i$. The first case contradicts the induction step, so suppose that $0 < \angle xvt < xvb_i$. Then for some $0 \leq j < i$, $t$ lies

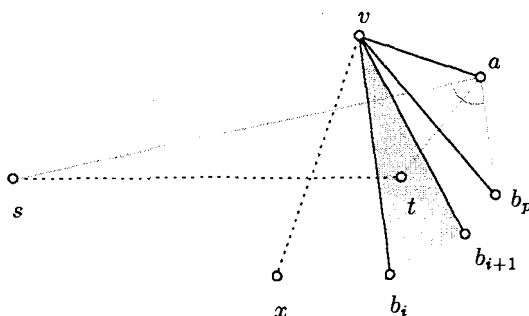


Figure 2.3: Invalid position for $t$ when $\mathcal{R}(v, x) = a$

within the convex hull of the vertices $v, b_j, b_{j+1}$, as shown in Figure 2.4. However, by Claim 2.2, this is impossible. □

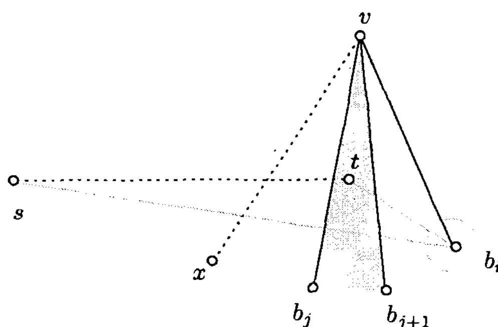


Figure 2.4: Invalid position for $t$ when $\mathcal{R}(v, x) = b_i$

**Claim 2.4** *Suppose $v \neq t$. Then $s \prec l_0 \prec \ldots \prec l_k \prec t$ where $\prec$ is the natural ordering along $st$.*

*Proof.* It follows from Claim 2.3 that $l_j$ is well-defined (*i.e.*, the intersection of $vx$ with $st$ exists) and that $s \prec l_j \prec t$ for all $0 \leq j \leq k$. We now assume for some $0 \leq j < k$ that $v \in V_A$; the case $v \in V_B$ is similar.

Since all underlying faces are convex, the angle between any radially adjacent edges is less than $\pi$. Therefore, the point of intersection of $st$ with $vb_i$ precedes that of $st$ with $vb_{i+1}$ for all $0 \leq i < p$, and the point of intersection of $st$ with $vb_p$ precedes that of $st$ with $b_pa$. Regardless of the choice of $\mathcal{R}(v, x)$, we must then have $l_j \prec l_{j+1}$. □

This concludes the proof of Theorem 2.2. ■

Observe that QUASI-PLANAR runs in polynomial time since the intersections $l_k$ are determined by the vertices $v$, $x$, $s$, and $t$, and there are at most $\binom{|V|}{2}$ choices for $v$ and $x$. Moreover, the algorithm only uses those underlying faces crossing the line segment $st$.

### 2.1.2 Rules for choosing the next vertex

Below we list several natural choices for the rule $\mathcal{R}$. The first three are greedy variants adapted for the QUASI-PLANAR algorithm; specifically, Rules 1, 2, and 3 are analogous to GREEDY, COMPASS, and GREEDY-COMPASS, respectively.

Rules 4 and 5 use network load information (assuming it is locally accessible) to choose vertices with minimum load, with the objective of keeping the maximum network load over all vertices as small as possible. Rule 4 chooses the vertex with least load from the entire set of candidates $\{load(b_1), \ldots, load(b_p), load(a)\}$. However, slavishly taking minimum-load vertices in this way typically produces very long and indirect $s,t$-paths; thus, a single message sent with Rule 4 may not significantly increase the maximum network load, but many such messages will cumulatively have a catastrophic effect on the network. Much better is the refinement in Rule 5, which restricts Rule 4 to the (GREEDY-COMPASS-like) choice of $b_p$ or $a$.

1. $\mathcal{R}(v,x) = argmin\{dist(b_1,t), \ldots, dist(b_p,t), dist(a,t)\}$
2. $\mathcal{R}(v,x) = argmin\{\angle b_p vt, \angle tva\}$
3. $\mathcal{R}(v,x) = argmin\{dist(b_p,t), dist(a,t)\}$
4. $\mathcal{R}(v,x) = argmin\{load(b_1), \ldots, load(b_p), load(a)\}$
5. $\mathcal{R}(v,x) = argmin\{load(b_p), load(a)\}$
6. $\mathcal{R}(v,x) = a$

The simplest choice, Rule 6, effectively ignores all $st$-crossing edges; it is easy to see that this rule is equivalent to FACE-2 showing that QUASI-PLANAR generalises FACE-2.

Obviously Rule 6 is a poor choice for $\mathcal{R}$ under most circumstances. However, it plays an integral rôle in the SPIRAL algorithm in Chapter 3, so we end this section with the following observation.

Since $v$ and $x$ are always on the same underlying face, and the QUASI-PLANAR algorithm progresses along $st$-crossing faces, it follows that at least one endpoint of every $st$-crossing edge in the underlying planar graph will be visited during the course of the algorithm. Moreover, if $v$ is at the endpoint of such an edge (say $v \in V_A$), then the other endpoint is either $x$ or in the set $\{a, b_1, \ldots, b_p\}$. Now, as written, the choice of next vertex never includes $x$, for the obvious reason that the algorithm should not cycle. However, when using Rule 6, note that we can modify the algorithm to include $x$ in the set of candidates – this is a trivial change since the rule will never choose $x$, but it allows us to say that the algorithm considers all $st$-crossing edges in the underlying planar graph when using Rule 6.

### 2.1.3 Analysis

Let us consider the behaviour of the QUASI-PLANAR algorithm in an idealised setting. Since QUASI-PLANAR only travels across underlying $st$-crossing faces, and never returns to a previously visited face, we might examine how QUASI-PLANAR performs on a single face to get an idea of the algorithm's performance on a general quasi-planar graph. In particular we will show that the behaviour of QUASI-PLANAR on a face is comparable to that of GREEDY and COMPASS. In this section we assume that QUASI-PLANAR is using Rule 1, *i.e.*, $\mathcal{R}(v, x) = argmin(dist(b_1, t), \ldots, dist(b_p, t), dist(a, t))$.

Let the underlying planar graph be a cycle $C$ whose vertices are embedded on a circle. Since scaling does not affect the behaviour of QUASI-PLANAR, GREEDY, or COMPASS, we may assume that the circle has diameter 1. Let $Q$ consist of this underlying graph along with an arbitrary number of chords.

We will now show that QUASI-PLANAR performs as effectively as GREEDY and COMPASS on $Q$. More precisely, all three algorithms follow the same $s, t$-path for all choices of $s$ and $t$.

**Theorem 2.3** *Let $Q$ be a quasi-planar graph as described above, and let $s, t \in V(Q)$. Then the* QUASI-PLANAR, GREEDY, *and* COMPASS *algorithms visit the same sequence of vertices when routing from $s$ to $t$.*

*Proof.* We will first prove that GREEDY and COMPASS visit the same sequence of vertices. Let $v$ be the current vertex, and let $u \in N(v)$. Let $\theta = \angle tvu$, and consider $\theta$ to be

an angle in the range $(-\pi, \pi)$. Observe that of the two neighbours of $v$ in the underlying graph $C$, the closer one to $t$ is also the one minimising angle to $t$, and that this angle is no greater than $\pi/2$. Therefore we may assume that $\theta \in (-\pi/2, \pi/2)$.

Since $u, v, t$ are on a circle of diameter 1, the line segment $ut$ has length $\sin\theta$. Therefore $u$ will be chosen by GREEDY as the next vertex if it minimises $|\sin\theta|$. But since $\theta \in (-\pi/2, \pi/2)$, minimising $|\sin\theta|$ is equivalent to minimising $|\theta|$, which is the criterion used by COMPASS. Finally, at least one of $v$'s neighbours is strictly closer to $t$ than $v$ is, so GREEDY and COMPASS must terminate.

We now prove that GREEDY and QUASI-PLANAR visit the same sequence of vertices. By choice of the rule, QUASI-PLANAR will select either $\mathbf{cw}(v,t)$ or $\mathbf{ccw}(v,t)$ as the next vertex. Let the closer of these to $t$ be called $u$. Clearly, GREEDY selects $u$ as the next vertex, so we must only show that QUASI-PLANAR does so as well.

Finally, it is clear from the choice of $\mathcal{R}$ that QUASI-PLANAR makes the same choices as GREEDY.

On the other hand, these three algorithms can have arbitrarily bad dilation on such graphs. Let the underlying cycle be $C = u_0u_1 \ldots u_{2n-1}u_0$, and add chords $u_0u_2$ and $u_1u_n$, as shown in Figure 2.5. Let $s = u_0$ and $t = u_n$. Then GREEDY clearly takes the path $P = u_0u_2u_3 \ldots u_n$, whereas the shortest $s,t$-path is $u_0u_1u_n$; the dilation of $P$ is $\frac{n-1}{2}$.

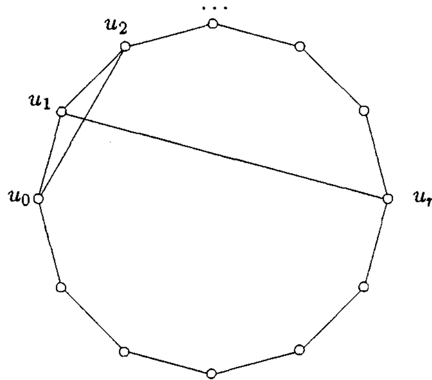


Figure 2.5: QUASI-PLANAR, GREEDY, and COMPASS can have arbitrarily bad dilation.

The behaviour of QUASI-PLANAR differs from that of GREEDY and COMPASS on more complex quasi-planar graphs, since QUASI-PLANAR is restricted to the faces intersecting $st$ and only moves to the next face when at an endpoint of an $st$-crossing

edge, while the latter algorithms ignore any inherent facial structure.

It is straightforward to construct quasi-planar graphs where GREEDY and COMPASS have arbitrarily better performance than QUASI-PLANAR, for example by drawing a path of length two between $s$ and $t$ whose edges are not incident to any $st$-crossing faces. On the other hand, in the introduction we saw that GREEDY and COMPASS do not even guarantee success on convex embeddings. Thus, in light of these examples and Theorem 2.3, we can expect the three algorithms QUASI-PLANAR, GREEDY, and COMPASS to be roughly comparable on "average" quasi-planar graphs, but to differ in some pathological cases.

## 2.2 QFQ: a hybrid algorithm for unit disk graphs

In Chapter 1 we described the GFG algorithm for unit disk graphs; recall that GFG alternates between GREEDY and face routing on the Gabriel subgraph, using FACE-1 only when GREEDY fails (*i.e.*, reaches a vertex $u$ whose distance to $t$ is lesser than all of its neighbours'), and reverting to GREEDY as soon as possible (*i.e.*, after finding a vertex $w$ such that $dist(w,t) < dist(u,t)$). We now show how to build a similar hybrid, QFQ, by combining QUASI-PLANAR with face routing on the Gabriel subgraph. For the remainder of this section we assume that the graph under consideration is a unit disk graph.

Clearly, any routing algorithm that does not cycle (but possibly terminates before reaching the destination) can be analogously combined with FACE-1 to produce a hybrid algorithm that guarantees delivery on unit disk graphs. Naturally we are interested in using QUASI-PLANAR as the base algorithm; hence we must first modify the algorithm to make it more robust.

Whereas GREEDY uses distance from $t$ as a measure of progress, QUASI-PLANAR uses the points of intersection $vx \cap st$, as we saw in the proof of Theorem 2.2. As written, QUASI-PLANAR does not explicitly determine these points, but since the necessary calculation uses only $v$,$x$,$s$, and $t$, we can carry out the calculation locally at every iteration to measure progress. Thus we will store a point $p$ on $st$ to this end: initially, $p = s$, and at every iteration, calculate $l_w = wx_w \cap st$ for every candidate $w$ for the next vertex, where $x_w$ is the position that would be stored as $x$ if $w$ were chosen. If no such $l_w$ satisfies $p \prec l_w \prec t$, then we consider QUASI-PLANAR to have failed, and switch

to face routing on the Gabriel subgraph. Otherwise, choose a vertex $w$ satisfying the given condition (the rule $\mathcal{R}$ can be extended to account for this choice), replace $p$ with $l_w$, move to $w$, and continue. Conversely, in face routing mode, at every iteration check whether some neighbour $w$ of $v$ is the endpoint of an $st$-crossing edge $vw$ and satisfies $p \prec l_w \prec t$, where $l_w = vw \cap st$. If so, choose such a vertex $w$ (again with an extension of $\mathcal{R}$), let $x = w$, update $p$, and revert to QUASI-PLANAR. [2]

We call the resulting hybrid algorithm QFQ. By design, the algorithm obviously guarantees delivery on both quasi-planar graphs and unit disk graphs. Of course, we may also use QFQ as a heuristic on graphs that are "almost" unit disk graphs – for example, faulty unit disk graphs (see Section 1.4.5). Moreover, it is reasonable to expect that QFQ outperforms GFG on such graphs when the faultiness $\epsilon$ is high: GREEDY is less robust than QFQ, switching into face routing mode more often, and the Gabriel subgraph becomes less predictable (and "less planar") as $\epsilon$ increases.

Indeed, experimental results seem to confirm this hypothesis. A full series of tests is planned for the journal version of this chapter.

## 2.3 Quasi-polyhedral routing in $\mathbb{R}^3$

In this section we extend the notion of quasi-planar graphs to *quasi-polyhedral graphs* in $\mathbb{R}^3$, and describe a routing algorithm on these graphs.

### 2.3.1 Quasi-polyhedral graphs

Let $V$ be a set of vertices in $\mathbb{R}^3$, not all coplanar, and let $P_O$ be the convex hull of $V$. Consider a geometric graph $G = (V, E)$. If the edges of $G$ determine a set of convex polyhedra such that any two polyhedra are either disjoint or intersect in exactly one vertex, edge, or face, and if moreover their union is $P_O$, then we say $G$ is a *polyhedral graph*. We use $\mathcal{P}$ to denote the set of these polyhedra along with $P_O$, and call $P_O$ the *outer polyhedron* of $G$. Note that $\mathcal{P}$ is not necessarily uniquely determined by $(V, E)$, but this is not important for our purposes.

[2]Incidentally, observe that we can use the same principle to extend QUASI-PLANAR for more general quasi-planar graphs where we relax the assumption that $f_O$ is convex. An agent can switch into a face routing mode on $f_O$ when necessary, and revert to the standard QUASI-PLANAR procedure after finding an $st$-crossing edge on $f_O$ that indicates progress towards $t$.

The intersection of any two polyhedra in $\mathcal{P}$ is either empty, or consists of a vertex, edge, or polygonal *face* in $G$. Let $F$ be the set of all faces determined by $\mathcal{P}$. We say $f \in F$ is a face of the polyhedron $P \in \mathcal{P}$ if $f \cap P = f$. A polyhedral graph $G$ may now be described by the 4-tuple $(V, E, F, \mathcal{P})$.

For three distinct, not necessarily adjacent vertices $a, b, c \in V$, denote by $\triangle abc$ the triangle with vertices $a, b, c$. A *3-cycle* $abc$ is a triple of pairwise adjacent vertices $a, b, c \in V$.

As in the previous section, we will assume that no three vertices are collinear. We similarly assume that no four vertices are coplanar, so that every face in $F$ is a triangle. Note, however, that not every 3-cycle is a face. For example, consider a polyhedral graph $G$ consisting of two tetrahedra with vertices $\{a, b, c, x\}$ and $\{a, b, c, z\}$ joined along the common face $abc$. Add a vertex $y$ in the interior of $G$ adjacent to all five existing vertices. The 3-cycle $abc$ is no longer a face of any polyhedron, since $\triangle abc$ intersects either $xy$ or $yz$. This is shown in Figure 2.6.

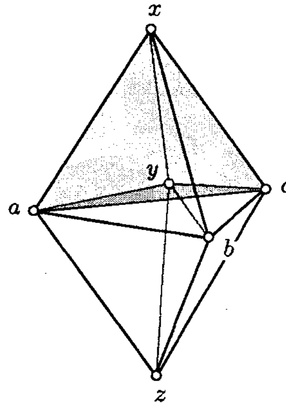


Figure 2.6: The pairwise adjacent vertices $a, b, c$ compose a 3-cycle, but not a face. There are six polyhedra in the graph: the tetrahedra with vertex sets $\{a, b, x, y\}$, $\{b, c, x, y\}$, $\{c, a, x, y\}$, $\{a, b, y, z\}$, $\{b, c, y, z\}$, and $\{c, a, y, z\}$. The polyhedron $\{c, a, x, y\}$ is shaded in the figure.

As an analogue of quasi-planar graphs, we now add chords to a polyhedral graph, so long as the chords join vertices on the same polyhedron (except the outer polyhedron $P_O$). That is, for some polyhedral graph $G = (V, E, F, \mathcal{P})$, construct $Q = (V, E \cup E', F, \mathcal{P})$, where each edge in $E'$ joins two vertices of a polyhedron $P \in \mathcal{P} \setminus \{P_O\}$. We say that $Q$ is a *quasi-polyhedral graph*, and that $G$ is an *underlying polyhedral graph* of $Q$ ($G$ is not

necessarily unique for $Q$). For brevity, we will usually use the term *polyhedron* rather than the more formal *underlying polyhedron*.

### 2.3.2 The QUASI-POLYHEDRAL algorithm

Similarly to the planar face-routing algorithms, QUASI-POLYHEDRAL travels only through polyhedra intersecting the line segment $st$. Whereas QUASI-PLANAR uses only one reference vertex $x$, QUASI-POLYHEDRAL stores two reference vertices $x$ and $y$, maintaining the properties that $v, x, y$ are on the same polyhedron $P$, and that $\triangle vxy$ intersects $st$.

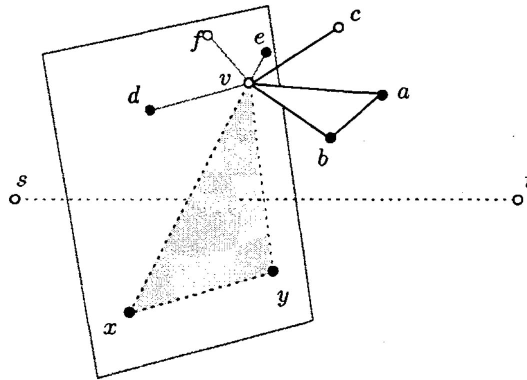


Figure 2.7: Candidates for the next vertex include $a$ and $b$, which are feasible and forward. The other neighbours of $v$ are not candidates since they are infeasible ($c$,$f$), backward ($d$,$e$,$f$), or both ($f$). The diagram also depicts the plane through $\triangle vxy$.

We will call a neighbour $u$ of $v$ *feasible* if there exists a polyhedron $P \in \mathcal{P}$ whose vertices include $v, x, y$, and $u$; otherwise $u$ is *infeasible*. A *feasible face* is a face whose vertices are all feasible. A face with at least one infeasible vertex is an *infeasible face*. Note that a feasible vertex can be a member of many infeasible faces. A vertex $u \in N(v)$ is said to be a *forward vertex* if $u$ is separated from $s$ by the plane through $\triangle vxy$. Otherwise, $u$ is a *backward vertex*. An example illustrating these definitions is depicted in Figure 2.7. To determine the first move from $s$ (*i.e.*, the first location of vertex $v$) and the initial reference vertices $x$ and $y$, QUASI-POLYHEDRAL uses a subroutine FIND FEASIBLE INITIALISATION (FFINIT). Then QUASI-POLYHEDRAL progresses towards $t$ in each iteration, using a subroutine FIND FORWARD FEASIBLE NEIGHBOUR (FFF) to choose the next vertex from the feasible forward neighbours of the current vertex $v$.

These subroutines are similar to the corresponding computations in QUASI-PLANAR; in particular, FFINIT is analogous to steps 2– 3 and FFF to steps 6 and 15. However, there is a subtle geometric complication that requires some explanation, so we delay the descriptions of these subroutines until Section 2.3.3.

**Algorithm 3** Quasi-Polyhedral Routing

```
1: procedure QUASI-POLYHEDRAL(Q, s, t, R)
2:     {v, x, y} ← FFINIT(Q, s, t)
3:     while vt ∉ E do
4:         w ← FFF(Q, s, t, v, x, y)
5:         if △wxy intersects st then
6:             v ← w
7:         else if △vwy intersects st then
8:             x ← y
9:             y ← v
10:            v ← w
11:        else                                  ▷ △vxw intersects st
12:            y ← x
13:            x ← v
14:            v ← w
15:        end if
16:    end while
17:    v ← t
18: end procedure
```

Assuming the correctness of FFF and FFINIT for now, we prove that QUASI-POLYHEDRAL (Algorithm 3) successfully routes on quasi-polyhedral graphs.

**Theorem 2.4** *Given a quasi-polyhedral graph $Q$ and distinct, non-adjacent vertices $s, t \in V(Q)$, the* QUASI-POLYHEDRAL *algorithm successfully routes from $s$ to $t$.*

*Proof.* The proof is structured analogously to the proof of Theorem 2.2. We will show that $v, x$, and $y$ are on the same underlying polyhedron during the execution of QUASI-POLYHEDRAL. Furthermore, let $l_k$ denote the point of intersection of $\triangle vxy$ with $st$ after the $k$th iteration of the `while` loop. We will also show that if $v \neq t$ after the $k$-th iteration, then $l_k$ exists and $s \prec l_0 \prec \ldots \prec l_k \prec t$ where $\prec$ is the natural ordering along $st$.

The intersection points $l_k$ are determined by triples of distinct vertices in $Q$, so the sequence $l_0, l_1, \ldots$ has at most $\binom{|V|}{3}$ terms. The `while` loop iterates as long as

$vt \notin E$, resulting in a new intersection point with each iteration. Therefore, since this sequence of points is finite, it follows that after some iteration, $vt \in E$. The `while` loop then terminates and $v$ reaches $t$ at step 17.

Therefore, it remains to prove the above two statements (Claims 2.5 and 2.7 in what follows). We proceed by induction on $k$, the number of iterations of the `while` loop in steps 3–16.

**Claim 2.5** *Vertices $v, x$, and $y$ are on the same underlying polyhedron.*

*Proof.* For $k = 0$, this follows from the choice of $x$ and $y$ from FFINIT in step 2. For $k \geq 1$, FFF finds a feasible vertex $w$ in step 4. By definition, $w$ is on the same polyhedron as $v, x$, and $y$. Steps 5–15 only permute the vertices $v, x$, and $y$, and one of them is assigned $w$. This maintains the desired property. □

**Claim 2.6** *The intersection point $l_k$ is well-defined,* i.e., *the triangle $\triangle vxy$ intersects the line segment $st$.*

*Proof.* Let $\ell$ be the line through $st$. We will first prove that $\triangle vxy$ intersects $\ell$, then use this to show that the point of intersection lies on the line segment $st$.

When $k = 0$, $\triangle vxy$ intersects $\ell$ at $s$ since $v = s$. For $k > 0$, suppose that $\triangle vxy$ intersected $\ell$ after the $k-1$st iteration of the `while` loop. Let $w$ be the vertex chosen by FFF in step 4 during the $k$-th iteration of the `while` loop. We will show that at least one of the triangles $\triangle wxy, \triangle vwy, \triangle vxw$ intersects $\ell$.

Project $V$ onto a plane $S$ perpendicular to $\ell$, denoting the image of a vertex $u$ by $\hat{u}$. Then the line $\ell$ is projected onto one point $\hat{s}$. The images $\hat{v}, \hat{x}$, and $\hat{y}$ are distinct since $\triangle vxy$ intersects $\ell$, and no four vertices are coplanar.

Let $C_{xy}$ be the reflection of $cone(\hat{x}, \hat{s}, \hat{y})$ through its axis of symmetry across $\hat{s}$, as shown in Figure 2.8. For any $u \in V$, it is clear that $\triangle uxy$ intersects $s$ if and only if $C_{xy}$ contains $\hat{u}$.

Define $C_{vx}$ and $C_{vy}$ similarly. Then $C_{xy} \cup C_{vx} \cup C_{vy} = S$, so at least one of $\triangle\hat{w}\hat{x}\hat{y}, \triangle\hat{v}\hat{w}\hat{y}, \triangle\hat{v}\hat{x}\hat{w}$ contains $\hat{s}$. Finally, a triangle intersects $\ell$ in the original graph if and only if its projection onto $S$ contains $\hat{s}$.

It follows from steps 5–15 that $\triangle vxy$ intersects $\ell$ at the end of the $k$-th iteration; call the point of intersection $l_k$. We now show that $l_k$ must lie on the line segment $st$.

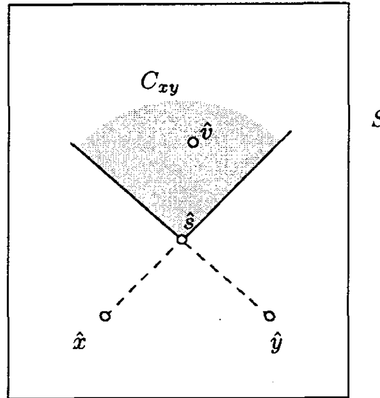


Figure 2.8: The cone $C_{xy}$.

Since $l_0 = s$, we can assume that $k > 0$ and that $l_{k-1}$ lies on $st$. Let $w$ be the vertex chosen by FFF in step 4. Then, since $w$ is a forward feasible neighbour of $v$, the vertices $v, x, y$, and $w$ lie on a polyhedron $P$; also, $w$ and $t$ are on the same side of the plane through $\triangle vxy$. Therefore, if $l_k$ does not lie on $st$, $t$ must be contained in $P$, a contradiction. It follows that one of $\triangle wxy, \triangle vwy, \triangle vxw$ intersects $st$, so $w$ will replace one of $v, x, y$ in steps 5–15 such that the desired property is maintained. □

**Claim 2.7** *Suppose $v \neq t$. Then $s \prec l_0 \prec \ldots \prec l_k \prec t$ where $\prec$ is the natural ordering along $st$.*

*Proof.* It follows from Claim 2.6 that $l_j$ is well-defined (*i.e.*, the intersection of $\triangle vxy$ with $st$ exists) and that $s \prec l_j \prec t$ for all $0 \leq j \leq k$.

Since all underlying polyhedra are convex, the angle between $\triangle wxy$ and $\triangle vxy$ is less than $\pi$. The same holds with respect to $\triangle vxy$ for triangles $\triangle vwy$ and $\triangle vxw$. Therefore, $l_j \prec l_{j+1}$ for all $0 \leq j < k$. □

This concludes the proof of Theorem 2.4. ■

QUASI-POLYHEDRAL runs in polynomial time: since the intersections $l_k$ are determined by the vertices $v$, $x$, $y$, $s$, and $t$, and there are at most $\binom{|V|}{3}$ choices for $v$, $x$, and $y$. Moreover, the algorithm only uses those underlying polyhedra properly intersecting the line segment $st$.

### 2.3.3 The FFINIT and FFF subroutines

In this section we describe both the FIND FEASIBLE INITIALISATION (FFINIT) and FIND FEASIBLE FORWARD NEIGHBOUR (FFF) algorithms and prove their correctness. First we need some definitions.

An *oriented plane* in $\mathbb{R}^3$ is a plane $S$ along with two spanning vectors a and b that play the rôles of the standard unit vectors $[10]^T$ and $[01]^T$, respectively, in $\mathbb{R}^2$. We say that $S$ has *orientation* $(\mathrm{a}, \mathrm{b})$. The orientation makes it possible to measure clockwise and counterclockwise angles on $S$.

Let $C = vab$ be a 3-cycle, and let $S$ be an oriented plane through $v$ intersecting $ab$ at some point $m$. Let $u$ be a point (not necessarily a vertex) on $S$. Then $\angle_S uvC$ denotes the counterclockwise angle $\angle uvm$ from $u$ to $m$ around $v$, as measured on $S$; similarly $\angle_S Cvu$ denotes the angle $\angle mvu$. See Figure 2.9.

This naturally suggests functions $\mathbf{ccw}_S(v, u)$ and $\mathbf{cw}_S(v, u)$ that return the 3-cycle $C$ minimising the non-zero angle $\angle_S uvC$, respectively $\angle_S Cvu$, such that $ab$ intersects $S$. Note that $C$ is not necessarily unique; for our purposes it is enough to choose a 3-cycle with minimal angle.

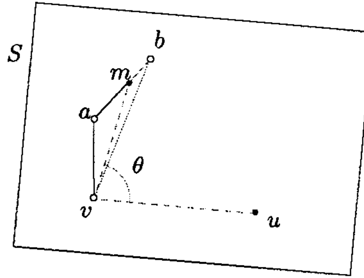


Figure 2.9: $S$ is an oriented plane through $v$, and intersecting $ab$; $u$ is a point on $S$. The angle on $S$ from $vu$ to the 3-cycle $vab$ is $\theta = \angle uvm$ where $m = ab \cap S$.

We will use these functions in FFINIT and FFF to find initial vertices $v, x, y$, and candidates for the next vertex, respectively. However, there is one issue to consider before implementing them. Recall Figure 2.6, which showed a non-facial 3-cycle. Observe that in this example, $yz$ intersects $\triangle abc$, while no edge intersects $\triangle ayz$, as shown in Figure 2.10(a). This indicates a means of identifying some of the "bad" 3-cycles in the graph.

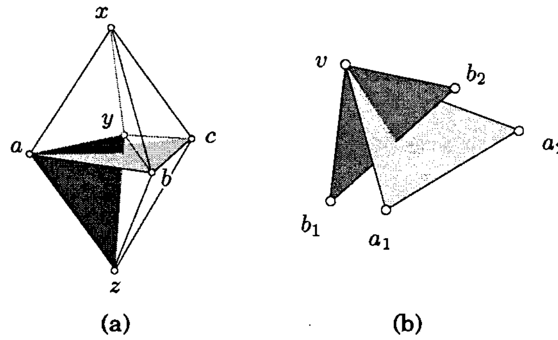


Figure 2.10: (a) The edge $yz$ of the 3-cycle $ayz$ intersects the triangle $\triangle abc$. (b) The 3-cycle $va_1a_2$ dominates $vb_1b_2$.

Let $C_a = va_1a_2$ and $C_b = vb_1b_2$ be 3-cycles, with $a_1, a_2, b_1, b_2$ distinct. Suppose that the line segment $b_1b_2$ intersects $\triangle va_1a_2$; see Figure 2.10(b). Then we say $C_a$ *dominates* $C_b$. If a 3-cycle $C$ dominates another 3-cycle, then $C$ is a *dominating* 3-cycle.

We will call the feasible faces incident to $v$ *cap faces*; see Figure 2.11. As we show in Lemma 2.5, it is impossible for a cap face to dominate another 3-cycle through $v$. This allows us to safely ignore all dominating 3-cycles.

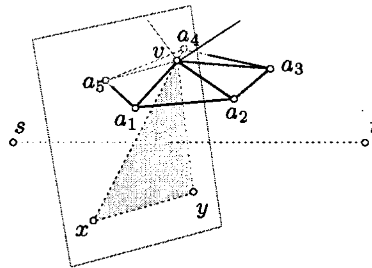


Figure 2.11: The five faces $va_1a_2, \ldots, va_5a_1$ are cap faces. The portions of the faces on the forward side of the plane through $\triangle vxy$ are lightly shaded, and the forward portions of the edges are bold.

**Lemma 2.5** *Let $f = va_1a_2$ be a face. Then $f$ does not dominate any other 3-cycle through $v$.*

*Proof.* Let $f$ be a face of some polyhedron $P \in \mathcal{P}$. Towards a contradiction suppose $f$ dominates a 3-cycle $vb_1b_2$. Then $b_1b_2$ intersects $f$. Therefore $b_1b_2$ intersects $P$, and by convexity of the polyhedra and definition of quasi-polyhedral graphs, at least one of the

$b_i$, say $b_1$, is contained in $P$. But since $b_1b_2$ intersects $f$, $b_2$ must be outside $P$. Since $P$ is convex and $b_1b_2$ intersects $f$, $b_1b_2 \notin E \cup E'$, a contradiction. □

**FIND FEASIBLE INITIALISATION (FFINIT)**

The FFINIT subroutine proceeds as follows. First, it chooses an arbitrary oriented plane $S$ through $st$. Then, by repeated application of $\mathbf{ccw}_S$, it finds the first non-dominating 3-cycle $sa_1a_2$ counterclockwise from $t$ around $s$. It then finds the first non-dominating 3-cycle $sb_1b_2$ clockwise from $t$ around $s$. The vertices $a_1, a_2, b_1, b_2$ lie on the same polyhedron, and three of them must form a triangle intersecting $st$. These three will be the initial assignments to $v, x$, and $y$.

**Algorithm 4** Find Feasible Initialisation

```
 1: procedure FFINIT(Q, s, t)
 2:     Let S be an oriented plane through st.
 3:     l ← s
 4:     repeat
 5:         sa₁a₂ ← ccw_S(s, l)
 6:         l ← a₁a₂ ∩ S
 7:     until sa₁a₂ is not a dominating cycle
 8:     l ← s
 9:     repeat
10:         sb₁b₂ ← cw_S(s, l)
11:         l ← b₁b₂ ∩ S
12:     until sb₁b₂ is not a dominating cycle
13:     return three of {a₁, a₂, b₁, b₂} that form a triangle intersecting st.
14: end procedure
```

**Theorem 2.6** *Let $P$ be the polyhedron through $s$ that intersects $st \setminus s$. The* FFINIT *algorithm returns three vertices lying on $P$, and the triangle formed by these vertices intersects $st$.*

*Proof.* Let $S$ be an oriented plane $S$ through $st$. For simplicity, we may assume that $S$ passes through no vertices other than $s$ and $t$, so that there are exactly two cap faces of $P$ that intersect $S \setminus s$. Call these cap faces $f_a$ and $f_b$, where $0 < \angle_S stsf_a < \pi$ and $0 < \angle_S f_bst < \pi$.

By Lemma 2.5, $f_a$ and $f_b$ do not dominate any 3-cycle through $s$, so both `repeat` loops terminate.

Let $C = sa_1a_2$ be the first non-dominating 3-cycle counterclockwise from $t$ around $s$, *i.e.*, the final 3-cycle determined by the `repeat` loop in steps 4–7. We will show that $a_1$ and $a_2$ lie on $P$.

If $C$ is a cap face, we are done; therefore, suppose that $C$ is not a cap face. It follows that $0 < \angle_S tsC < \angle_S tsf_a$. Then, since $P$ intersects $st \setminus s$ and $f_a \setminus s$, and $P$ is convex, $P$ intersects $\triangle C \setminus s$. Now, towards a contradiction, suppose that $a_1$ does not lie on $P$. Since $P$ intersects $\triangle C \setminus s$, and $a_1$ is not on $P$ and lies on $C$, $\triangle C \setminus s$ must intersect some cap face $f$. By Lemma 2.5, $f$ does not dominate $C$, so $C$ must dominate $f$. But this is impossible by choice of $C$. Therefore $a_1$ must lie on $P$. The same reasoning shows that $a_2$, $b_1$ and $b_2$ lie on $P$.

We now show that (at least) one of the four triangles $\triangle a_1a_2b_1$, $\triangle a_1a_2b_2$, $\triangle a_1b_1b_2$, $\triangle a_2b_1b_2$ intersects $st$. Project $V$ onto a plane $T$ perpendicular to $st$, denoting the image of a vertex $u$ by $\hat{u}$. Then $st$ is projected onto one point $\hat{s}$, and the plane $S$ is projected onto a line $\hat{S}$. Both $\hat{a_1}\hat{a_2}$ and $\hat{b_1}\hat{b_2}$ intersect $\hat{S}$, and the points of intersection lie on different sides of $\hat{s}$. Therefore, $\hat{s}$ is in the interior of the quadrilateral with vertices $\hat{a_1}, \hat{a_2}, \hat{b_1}, \hat{b_2}$, so one of the four triangles $\triangle\hat{a_1}\hat{a_2}\hat{b_1}, \triangle\hat{a_1}\hat{a_2}\hat{b_2}, \triangle\hat{a_1}\hat{b_1}\hat{b_2}, \triangle\hat{a_2}\hat{b_1}\hat{b_2}$ contains $\hat{s}$. Since $P$ is convex and intersects $st \setminus s$, the corresponding vertices in $V$ form a triangle intersecting the line through $st$, and in particular, the point of intersection must lie on the line segment $st$. ■

**FIND FEASIBLE FORWARD NEIGHBOUR (FFF)**

Suppose $v, x$, and $y$ are on the same polyhedron $P$, and that $\triangle vxy$ intersects $st$. To find a feasible forward neighbour of $v$, FFF uses the same technique as FFINIT: it finds the first non-dominating 3-cycle through $v$ counterclockwise from $\triangle vxy$ and returns one of the endpoints.

**Theorem 2.7** *The* FFF *algorithm finds a forward feasible neighbour of* $v$.

*Proof.* First, the point of intersection $l = \triangle vxy \cap st$ is well-defined by the above assumption on the vertices $v, x, y$. Let $P$ be the polyhedron through $v, x$, and $y$; if these vertices lie on two polyhedra, then consider $P$ to be the one whose intersection with $st$ is closer to $t$. We can therefore imagine $\mathbf{ccw}_S$ to be sweeping through the interior of $P$ to find successive 3-cycles.

**Algorithm 5** Find Forward Feasible Neighbour

```
1: procedure FFF(Q, s, t, v, x, y)
2:     Let l = △vxy ∩ st.
3:     Let S be the plane through v, l, and t, with orientation (lt, lv).
4:     repeat
5:         va1a2 ← ccw_S(v, l)
6:         l ← a1a2 ∩ S
7:     until va1a2 is not a dominating cycle
8:     return a forward vertex from {a1, a2}.
9: end procedure
```

The `repeat` loop in steps 4–7 terminates, since a cap face of $P$ is valid by Lemma 2.5. Let $C = va_1a_2$ be the 3-cycle determined by the `repeat` loop. The same methods as in the proof of Theorem 2.6 show that $a_1$ and $a_2$ must lie on $P$.

Finally, by convexity of $P$ and the choice of orientation of $S$, $0 < \angle_S lvC < \pi$, so $(C \setminus v) \cap S$ lies in the forward region, *i.e.*, every point of $(C \setminus v) \cap S$ is separated from $s$ by $\triangle vxy$. Therefore, since $a_1a_2 \subset (C \setminus v)$, $a_1$ and $a_2$ cannot both be backward vertices. ■

Note that both FFINIT and FFF run in polynomial time, and only use $v, x, y, s$, and $t$ for their computations.

# Chapter 3

# Quasi-planar geocasting

## 3.1 The SPIRAL geocasting algorithm

In this section we present an algorithm, SPIRAL, that visits each vertex in a specified circular region of a quasi-planar graph $Q$ at least once. The algorithm can easily be modified to search any convex region whose boundary can be determined with $O(1)$ information.

Let $Q = (V, E \cup E')$ be a quasi-planar graph and let $G = (V, E, F)$ be an underlying planar graph for $Q$. As in Chapter 2, $G$ is not provided to or calculated by the agent running the algorithm; it is used only in the proofs.

Let $D(c, r)$ be the open disk centred at $c$ and with radius $r$; when $c$ and $r$ are understood we will simply write $D$. We denote the boundary of $D$, *i.e.*, the circle centred at $c$ with radius $r$, by $\partial D$. An edge $uv$ is said to be a *boundary edge* (with respect to $D$) if it intersects $\partial D$, or if it is an edge of $f_O$ and is contained in $D$. Note that the endpoints of a boundary edge may both lie outside $D$. An *underlying* boundary edge is a boundary edge in $E$. A face $f \in F$ is an *underlying boundary face* if it contains at least one boundary edge; we say that $f_O$ is also an underlying boundary face in the special case when every vertex in the graph is contained in $D$. For brevity we will write only *boundary face*, omitting the qualifying term "underlying", since faces are defined only on the underlying graph $G$. Finally, a vertex $v$ is a *boundary vertex* if it is the endpoint of a boundary edge; $v$ is an *inner* boundary vertex if it is inside $D$, and an *outer* boundary vertex if it is outside $D$ or on $f_O$. (Thus, a vertex is on $f_O$ and inside

$D$ if and only if it is both an inner and outer boundary vertex.) Figure 3.1 illustrates these definitions.

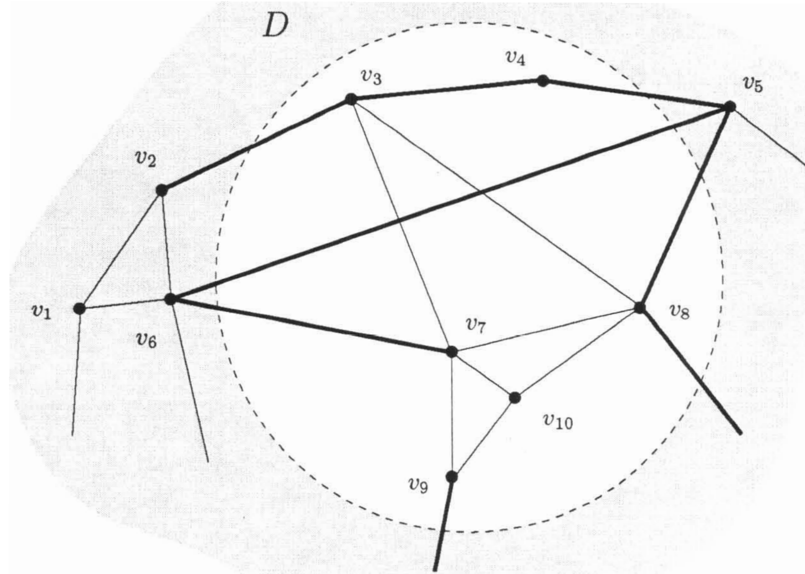


Figure 3.1: The boundary edges are drawn with thick lines. Note that $v_3v_4$ is a boundary edge since it lies on $f_O$, and that both endpoints of $v_5v_6$ are outside $D$. The (underlying) boundary faces include $f_O$ and the face with vertex set $\{v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$, but not those with vertex sets $\{v_1, v_2, v_6\}$ or $\{v_7, v_8, v_{10}\}$. Note that there is a unique underlying planar graph in this example. Since $v_3$ and $v_4$ lie on $f_O$ and are contained in $D$, they are both inner and outer boundary vertices.

We have the following result directly from the definitions.

**Lemma 3.1** *Suppose there exists a vertex $v$ inside $D = D(c, r)$. Then there exists at least one underlying boundary edge $xy$ such that $x$ and $y$ are inner and outer boundary vertices, respectively.*

*Proof.* If all vertices are contained in $D$, then every edge on $f_O$ is an underlying boundary edge, and every vertex on $f_O$ is both an inner and outer boundary vertex. Therefore suppose that there exists a vertex $w$ outside $D$. Consider any $v, w$-path $P$ in the underlying graph. Since $dist(v, c) < r$ and $dist(w, c) \geq r$, there obviously must exist an edge $xy$ on $P$ for which $dist(x, c) < r$ but $dist(y, c) \geq r$. This is the required edge. ∎

We now proceed with a description of the SPIRAL algorithm (Algorithm 6). Broadly speaking, the algorithm travels around $D$, visiting some of the inner boundary vertices. At the end of the main `while` loop (lines 2-21 in Algorithm 6), $D$ is replaced with a strictly smaller disk $D'$ and the algorithm continues on $D'$, unless $D'$ is empty (in other words, if it has negative radius). The agent stores a variable $r'$ representing the radius of $D'$, which is updated periodically; initially $r' = -1$, and the algorithm terminates if $r'$ is still $-1$ at the end of the main loop. (Note that the algorithm continues in the case when $r = 0$ since there may be a vertex at $c$.)

At the beginning of each iteration of the main loop, the algorithm calls a subroutine, QP-FINDBOUNDARY (a modification of the QUASI-PLANAR algorithm described in Chapter 2), to find an outer boundary vertex $s$. Then, starting from $s$, the agent navigates the perimeter of $D$ using QP-PERIMETER, another modification of QUASI-PLANAR. As with the standard QUASI-PLANAR algorithm, QP-PERIMETER uses a reference vertex $x$ to determine a set of candidates for choosing the next vertex. At every iteration during its navigation, QP-PERIMETER specifies a set $I(v, x)$ of neighbours of the current vertex $v$ which the agent visits in turn before proceeding around $D$. In the case where $v$ is on $f_O$, $I(v, x) = \{v\}$, and the visit is trivial. During a visit to a vertex $u \in I(v, x)$, the algorithm updates $r'$ according to the neighbours of $u$. Specifically, for each $w \in N(u)$ inside $D$, it replaces $r'$ with $\max\{r', dist(w, c)\}$.

QP-FINDBOUNDARY and QP-PERIMETER are discussed in more detail in Sections 3.1.1 and 3.1.2, below.

The next lemma is the core argument in the proof of correctness of the SPIRAL algorithm: it shows that we can use convexity to predict, to some extent, the positions of the vertices inside $D$.

**Lemma 3.2** *Let $D = D(c, r)$ be given. Let $J$ be the set of inner boundary vertices incident to underlying boundary edges, and let $W$ be the set of neighbours of $J$ inside $D$,* i.e., *$W = N(J) \cap D$. Let $u \in V \setminus J$ be contained in $D$. Then $u \in conv(W)$.*

*Proof.* Towards a contradiction suppose there exists a vertex in $V \setminus J$ inside $D \setminus conv(W)$. Take $u$ to be the farthest such vertex from the set $conv(W)$; that is, take such a vertex $u$ maximising $dist(u, conv(W))$.

Since $u \notin J$, it follows that $u$ is not on $f_O$. Therefore, by convexity of the underlying faces, for any specified open half-plane $H$ whose boundary intersects $u$, $u$ must have

a neighbour $w$ contained in $H$ such that $uw$ is an edge in the underlying graph. In particular, let $L$ be the shortest line segment from $u$ to $conv(W)$, and let $H$ be the open half-plane not containing $conv(W)$ whose boundary is perpendicular to $L$ and intersects $u$. See Figure 3.2. Let $w$ be a vertex in $H$ such that $uw \in E$. Suppose that $w$ is inside $D$. Then $w$ cannot be in $V \setminus J$, for by convexity and boundedness of $conv(W)$, we have $dist(w, conv(W)) > dist(u, conv(W))$, but $u$ was chosen to be the farthest vertex in $V \setminus J$ from $conv(W)$ inside $D$. But neither can $w$ be in $J$, for then $u$ would be a neighbour of a vertex in $J$, in which case $u \in W \subset conv(W)$. It follows that $w$ is outside $D$, but this is also impossible for otherwise $uw$ would be an underlying boundary edge, implying $u \in J$. ∎

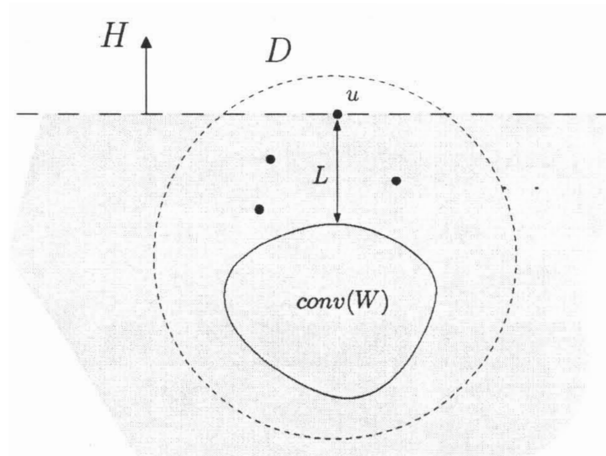


Figure 3.2: Proof of Lemma 3.2. The vertex $u \in V \setminus J$ is the farthest such vertex from $conv(W)$. $H$ is the open half-plane above the horizontal line.

**Theorem 3.3** *The* SPIRAL *algorithm finds all vertices inside the specified disk* $D(c, r)$.

*Proof.* We first mention that the statements of several of the lemmas used in this proof appear in the sections below. Let $D = D(c, r)$. First suppose that there do not exist any boundary edges with respect to $D$. Then by Lemma 3.1, $D$ contains no vertices. By Lemma 3.4, below, QP-FINDBOUNDARY will correctly report that there do not exist any outer boundary vertices, and SPIRAL terminates.

**Algorithm 6** Spiral geocasting

```
 1: procedure SPIRAL(Q, c, r)
 2:     while r ≥ 0 do
 3:         r' ← −1
 4:         Call QP-FINDBOUNDARY on D(c, r)
 5:         if QP-FINDBOUNDARY finds an outer boundary vertex s then
 6:             Navigate the perimeter of D(c, r) starting from s, using QP-PERIMETER
 7:             while QP-PERIMETER has not terminated do
 8:                 Let the current vertex be v and reference vertex be x.
 9:                 for every vertex u ∈ I(v, x) do
10:                     Move to u
11:                     for every vertex w ∈ N(u) inside D(c, r) do
12:                         r' ← max{r', dist(w, c)}
13:                     end for
14:                     Return to v
15:                 end for
16:             end while
17:         else
18:             Terminate
19:         end if
20:         r ← r'
21:     end while
22: end procedure
```

Now suppose that there exists a boundary edge. By Lemma 3.4, QP-FINDBOUNDARY will find an outer boundary vertex $s$; SPIRAL then searches the perimeter of $D$ according to QP-PERIMETER. Let $I = \bigcup_{(v,x)} I(v,x)$ and $W = \bigcup_{(v,x)} \bigcup_{u \in I(v,x)} N(u) \cap D$, where the (outer) unions are taken over all iterations during QP-PERIMETER's navigation, *i.e.*, during the `while` loop in lines 7-16 in Algorithm 6.

Let $J$ be the set of all inner boundary vertices incident to underlying boundary edges. By Lemma 3.8, $J \subseteq I$. SPIRAL visits all vertices in $I$ during the loop in lines 7-16; it is enough to know that it visits every vertex in $J$. Moreover, by Lemma 3.6, SPIRAL also visits every vertex on $\partial D$.

It remains to show that SPIRAL visits every other vertex in $D$ during subsequent iterations of the main loop, and that the algorithm terminates. Now, by Lemma 3.2, we know that every unvisited vertex in $D$ is contained in $conv(W)$. If $conv(W) = \emptyset$ then every vertex in $D$ has been visited. The algorithm terminates correctly since $r'$ is not updated during the main `while` loop, so at end of the loop, $r$ updates to $-1$; the algorithm will terminate at the beginning of the next iteration. Evidently, if $W$ is

nonempty, then at the end of the main loop, $r$ will be updated to $r^* = \max_{w \in W} dist(w, c)$. Since all vertices in $W$ are contained in $D$, obviously $r^* < r$. The new disk $D(c, r^*)$ contains all remaining vertices in $D$ since it clearly contains all vertices in $W$, and is a convex set so $D(c, r^*) \supset conv(W)$.

Finally, (at least) one vertex will be on the boundary of the new disk $D(c, r^*)$; therefore the number of iterations is bounded by the number of vertices in the original disk. Therefore the SPIRAL algorithm terminates; the running time is polynomial by Lemmas 3.5 and 3.7.

### 3.1.1 The QP-FINDBOUNDARY subroutine

In this section we show how to modify the QUASI-PLANAR algorithm to find an outer boundary vertex, starting from any vertex $s$. Obviously we can choose the destination $t$ to be a point rather than a vertex without significantly affecting the behaviour of the algorithm. If some vertex has position $t$, then the algorithm will find that vertex. Otherwise, either $t$ is an interior point of an underlying face, or lies outside the graph.

In the first case, if $t$ is an interior point of some edge, the algorithm will find one of the endpoints of the edge. Otherwise, the agent will eventually reach an edge crossing the line through $s$ and $t$, but not the line segment $st$; at this point the agent must be on an underlying face containing $t$. Finally, if $t$ is outside the graph, the agent will obviously reach the outer face, at which point it can terminate with failure.

We use this modified version of QUASI-PLANAR to find an outer boundary vertex when the agent is outside $D = D(c, r)$. Choose $c$ as the destination and run the (modified) QUASI-PLANAR algorithm, stopping if at any time the current vertex is a boundary vertex. If $D$ is strictly contained in some face and no chords intersect $D$, then the agent can naturally identify (part of) the underlying face containing $D$, and report that $D$ contains no vertices. Otherwise, either $D$ intersects some edge, or $D$ is outside the polygon determined by $f_O$; these cases are both handled as suggested in the previous paragraph.

On the other hand, if the agent is already inside $D$, then it can simply move to a vertex farther from $c$ than its current position is. If no such vertex exists, it must be on $f_O$, in which case we are done. Otherwise, it must clearly emerge from $D$ after a finite number of steps; when it first emerges from $D$ it must be on an outer boundary vertex.

Compare the reasoning here to the proof of Lemma 3.1.

We call this algorithm QP-FINDBOUNDARY; in summary, we state the following.

**Lemma 3.4** *Let $D = D(c, r)$. Starting from any initial vertex, the* QP-FINDBOUNDARY *algorithm either finds an outer boundary vertex, or reports that none exist. In the latter case $D$ contains no vertices.*

**Lemma 3.5** *The* QP-FINDBOUNDARY *algorithm runs in polynomial time.*

### 3.1.2 The QP-PERIMETER subroutine

Starting from an outer boundary vertex, an agent can navigate the perimeter of $D$ along the boundary faces by using a modified version of the QUASI-PLANAR algorithm with Rule 6 (see Section 2.1.2). We will describe the modifications in detail but justify them without formal proofs of correctness, since the arguments are very similar to those in Chapter 2.

Recall that an agent routing between some pair of vertices $(s, t)$ using QUASI-PLANAR with Rule 6 visits only those vertices above $st$ and remains within the $st$-crossing faces. In our current setting, the boundary $\partial D$ is analogous to $st$, with $V_A$ and $V_B$ corresponding to $\overline{D} := \mathbb{R}^2 \setminus D$ and $D$, respectively. In Chapter 2 we assumed for convenience that no vertices except $s$ and $t$ were on the line segment $st$; here, in contrast, there will be at least one vertex on $\partial D$ in nearly every iteration of SPIRAL (thus in nearly every instance of QP-PERIMETER), but this will not cause any problems. We will associate the (direction of the) directed line segment $st$ with the clockwise orientation of $\partial D$; the analogues of $s$ and $t$ in particular (*i.e.*, initialisation and terminating conditions) will be discussed soon.

It should be evident that in general we can analogously send an agent clockwise around the perimeter of the disk $D$ through the boundary faces, essentially by repeatedly applying the left-hand rule and skipping any edges intersecting $\partial D$ (*i.e.*, boundary edges). There is one important exception to this behaviour. Suppose that the current vertex $v$ is on $f_O$ and outside $D$, and that the next candidate edge $e$ incident to $v$ intersects $\partial D$ but is on $f_O$. Then the agent should take $e$ rather than skipping it, and travel along $f_O$ until emerging from $D$; note that the agent necessarily visits every vertex of $f_O$ inside $D$. (In the special case where every vertex is contained in $D$, the agent simply

traverses $f_O$.) Conversely, the agent is forced inside $D$ only on edges of $f_O$ intersecting $\partial D$. Figure 3.3 shows an example of the agent's walk around the disk.

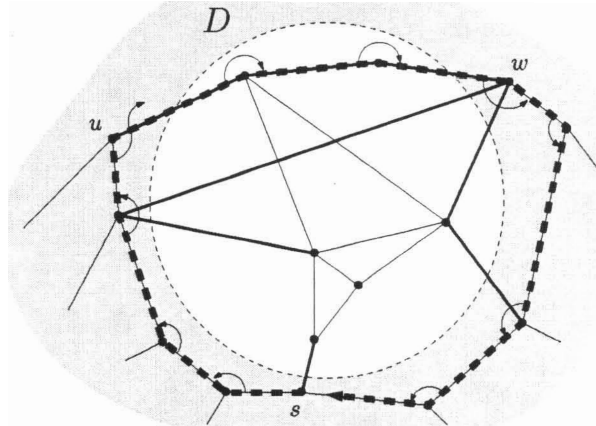


Figure 3.3: An agent starting from $s$ navigates the perimeter of the disk using the left-hand rule. It skips all boundary edges until it arrives at $u$, then traverses $f_O$ until emerging from $D$ at $w$. Then it resumes using the left-hand rule.

The reference vertex $x$ is handled the same way as in QUASI-PLANAR, with two minor changes: first, in the case where a boundary edge not on $f_O$ has both endpoints outside $D$ we simply ignore the edge for purposes of updating $x$; second, when the agent enters $D$ along $f_O$ we ignore $x$ until the agent emerges from $D$, at which point $x$ is set to the previous vertex (thus $x$ is always in $D$). See Figure 3.4.

Observe that a face may have arbitrarily many boundary edges, so an agent may return to that face many times during the routing, as demonstrated in Figure 3.5. For such a face $f$, consider the subgraph $Q_f$ of $Q$ consisting of all edges of $f$ that lie entirely outside $D$ (or, if $f = f_O$, all edges inside $D$). Then, from a local point of view, the connected components of $Q_f$ effectively belong to different faces. We can assign multiple labels to $f$ according to the components, so that when we say that an agent passes through all the boundary faces in order around $D$, the statement is understood in this local context.

We now discuss the algorithm's initialisation and terminating conditions. Let $s$ be the initial outer boundary vertex. If $s$ is on $f_O$, the agent begins by travelling clockwise around $f_O$; clearly it returns to $s$ when and only when it has passed through every

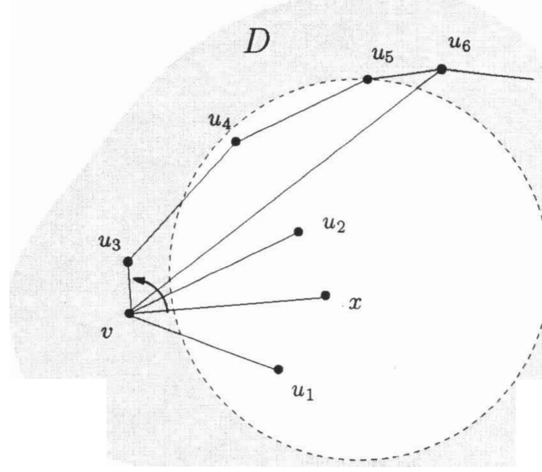


Figure 3.4: The current and reference vertices are $v$ and $x$, respectively. The set of candidates for the next vertex is $\{x, u_2, u_3\}$; contrast with Figure 2.2. Note that $u_6$ is not a candidate. In several iterations, the agent will enter $D$ along $f_O$. When it exits the disk at $u_5$ it will update $x \leftarrow u_4$. Note that the agent exits $D$ at $u_5$ rather than $u_6$, since $u_5$ is not strictly contained in $D$.

boundary face. Otherwise $s$ is outside $D$ and not on $f_O$. Choose a neighbour $w \in N(s)$ such that $sw$ is a boundary edge. If $w$ is inside $D$, then let $t$ be the point $sw \cap \partial D$; otherwise, of the two points in $sw \cap \partial D$, let $t$ be the closer to $s$. Note that $t = s$ if and only if $s$ is on $\partial D$. Also set the initial reference point $x$ to be $w$ if $w$ is inside $D$, or some point on $sw$ in the interior of $D$ otherwise (say the midpoint of the chord $sw \cap D$). It is easy to check that these initial settings for $x$ are appropriate, *i.e.*, that they are analogously consistent with those in QUASI-PLANAR.

In the proof of correctness of the QUASI-PLANAR algorithm, we measured progress towards (a vertex) $t$ according to the successive points of intersection of $vx$ with $st$; similarly we can measure progress towards (the point) $t$ around $D$ by calculating the points of intersection $vx \cap \partial D$. Evidently, after passing through every boundary face, the agent will eventually arrive at $t$ (which must be the vertex $s$) or a vertex $v$ such that $t$ is not on the line segment $v'x$ but is contained in $cone(v', x, v)$ where $v'$ is the next vertex that would be visited in the face routing; the algorithm then terminates. In other words, taking $\prec$ to be the natural clockwise ordering on $\partial D$ in a neighbourhood of $t$, we have $vx \cap \partial D \preceq t \prec v'x \cap \partial D$; compare this to Claim 2.4 in the proof of Theorem 2.2. Note

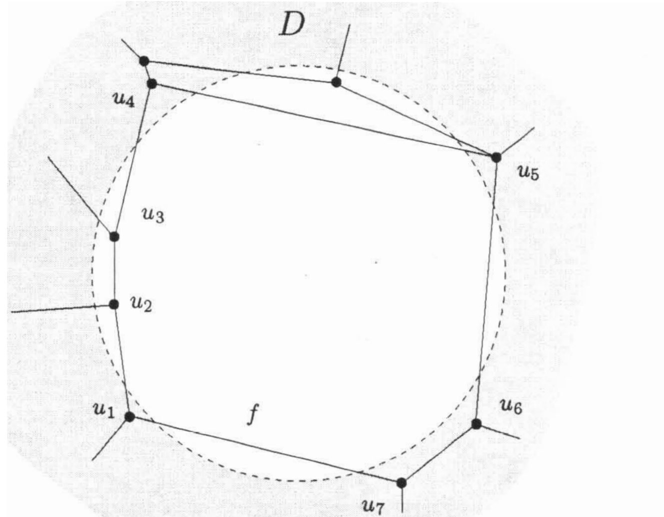


Figure 3.5: The boundary face $f$ with vertices $\{u_1, \ldots, u_7\}$ has eight boundary edges and will be visited several times by the agent. From a local point of view, however, we can consider $f$ to contribute distinct boundary faces with vertex sets $\{u_1\}$, $\{u_4\}$, $\{u_5\}$, and $\{u_6, u_7\}$.

that the terminating condition is a computation involving only the local positions $v$ and $v'$, the stored values $x$ and $t$, and the boundary $\partial D$ (determined by the stored values $c$ and $r$). Figure 3.6 illustrates the terminating condition. Finally, although it may be tempting to simplify the terminating condition by replacing it with the condition $v = s$ (*i.e.*, terminate after returning to the starting point), such a simplification is incorrect: the vertex $s$ may not be reachable in the face routing; this is also shown in Figure 3.6.

This completes the discussion of the QP-PERIMETER algorithm *per se*; we now show how to use it to visit all inner boundary vertices incident to an underlying boundary edge.

In Section 2.1.2 we noted that an agent routing with QUASI-PLANAR using Rule 6 considers every $st$-crossing edge in the underlying planar graph as a candidate at some time during the running of the algorithm. It follows from this, and from the observation that $f_O$ is a face in every underlying graph, that the QP-PERIMETER algorithm similarly considers every underlying boundary edge throughout the course of the navigation around $D$. Therefore, when the current vertex $v$ is outside $D$, it can specify a

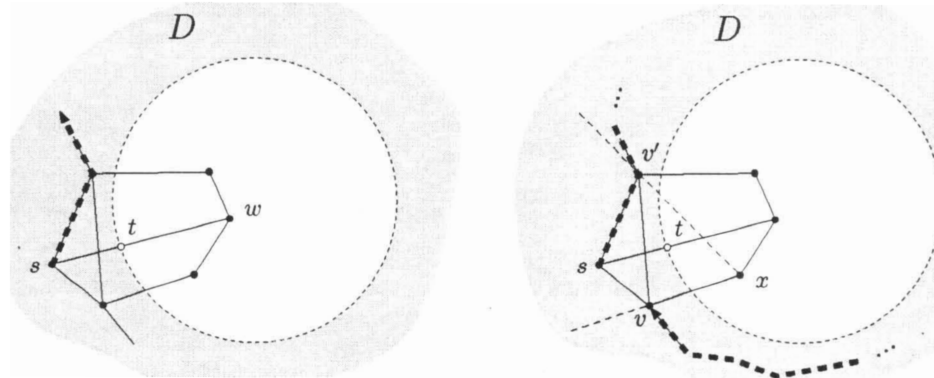


Figure 3.6: (a) From the initial vertex $s$ with inner boundary neighbour $w$, set $t \leftarrow sw \cap \partial D$. (b) Eventually the agent completes the navigation of $D$. The current vertex is $v$; the next vertex in the face routing would be $v'$, but the algorithm terminates since $t$ is contained in $cone(v', x, v)$. Note that if allowed to continue, the agent would never again visit $s$.

set $I(v, x)$ of vertices inside $D$ (including $x$ if $vx \in E \cup E'$) that includes all inner boundary vertices adjacent to $v$ through an underlying boundary edge. When $v$ is outside $D$, $I(v, x)$ is naturally the set of candidates considered for the next vertex, including $x$, but excluding any candidate outside $D$. For example in Figure 3.4, above, we have $I(v, x) = \{x, u_2\}$. If $v$ is on $f_O$ and the next vertex $v'$ is on $f_O$ but inside $D$, we need not include $v'$ in $I(v, x)$; it will be handled when the agent moves to $v'$. When $v$ is inside $D$, it is on $f_O$ and is already on an inner boundary vertex incident to an underlying boundary edge; conversely, every vertex of $f_O$ inside $D$ is visited during the running of the QP-PERIMETER algorithm. At such a vertex $v$ we define $I(v, x) = \{v\}$ for notational convenience. The SPIRAL algorithm calls for the agent to visit each vertex of $I(v, x)$ in turn; in the case when $v$ is already inside $D$ we consider these visits to be the trivial walks of length 0. Thus we can guarantee that the agent visits all inner boundary vertices incident to underlying boundary edges (and possibly other inner boundary vertices).

We summarise this section with the following lemmas, which are justified by the discussion thus far.

**Lemma 3.6** *Starting from an outer boundary vertex either on $f_O$ or incident to a vertex in D, the* QP-PERIMETER *algorithm passes through each boundary face (in a local*

*sense, as described above) no more than once, except possibly for some repeated vertices in the initial boundary face. In particular,* QP-PERIMETER *visits every vertex on $\partial D$.* □

**Lemma 3.7** QP-PERIMETER *runs in polynomial time.* □

**Lemma 3.8** *Let $u$ be an inner boundary vertex incident to an underlying boundary edge. Then $u \in I(v, x)$ for some pair $(v, x)$ during the course of the* QP-PERIMETER *algorithm.* □

# Chapter 4

# Disjoint Routing in Convex Embeddings

## 4.1 Overview

In this chapter we propose an extension of the standard routing problem discussed in Chapter 1. Let $G$ be a convex embedding with no three collinear vertices. As usual, a source node $s$ and a destination node $t$ are specified. We now consider the problem of finding multiple internally vertex-disjoint $s,t$-walks in $G$. For brevity we will often simply write "disjoint $s,t$-walks".

The ability to find disjoint $s,t$-walks in a communications network provides a number of benefits. For example, sensitive information can be encrypted and partitioned into several components in such a way that the original message can only be retrieved if all components are known. The components can then be sent to the destination along $k$ disjoint routes, so that an adversary attempting to intercept and decrypt the data must expend more resources by being required to compromise at least $k$ nodes. Or, a network may suffer from unreliable communication (*e.g.*, data corruption) between certain nodes; sending identical messages along disjoint routes adds redundancy at the expense of network load. As a final example, the source vertex $s$ may incur a cost for each edge used by a message originating from $s$. If the cost is highly nonlinear with respect to message size, then the total cost can be reduced by partitioning the message and sending the components along disjoint routes.

In Chapter 1 we saw that routing on a convex embedding is accomplished by using the left- or right-hand-rule along the upper or lower halves of $st$-crossing faces, respectively. This immediately suggests a method for finding two disjoint $s,t$-walks: send one message to $t$ using the left-hand-rule, and the other using the right-hand-rule, as demonstrated in Figure 4.1. It is clear from the convexity of the faces that the walks are disjoint, since neither walk uses $st$-crossing edges.

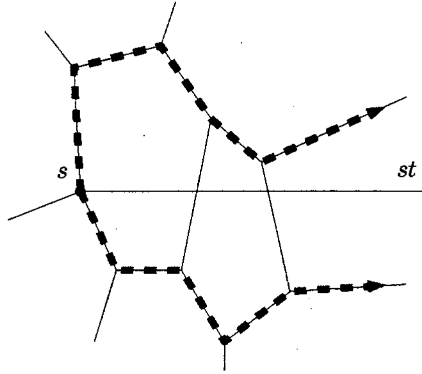


Figure 4.1: Two internally disjoint $s,t$-walks are determined by traversing the upper and lower halves of $st$-crossing faces.

As an aside, note that finding even two disjoint routes in a planar graph with non-convex faces is a much harder problem. Consider Figure 4.2, for example. Of the two chains of 4-cycles extending towards $t$, suppose that only one reaches $t$ and that the other is a dead end. (Note that unlike the other figures in this chapter, where most faces are omitted for clarity, here the graph consists only of the visible faces and those implied by ellipses). One message must pass through the vertex labelled $x$, but we cannot know which one without first searching the entire graph. In our algorithms we wish to bound any necessary searches.

In this chapter we will push one step further for convex embeddings, presenting an algorithm for finding three disjoint $s,t$-walks in such graphs. The algorithm, DOUBLE-CROSSING, can be considered an extension of the method described above for finding two disjoint walks, but handles special cases that do not arise in the latter problem.

For the remainder of this chapter we will naturally assume that $G$ is 3-connected. In contrast to Chapter 2, we now take $uv$ to denote the line segment joining $u$ and $v$, and not the line through those points.

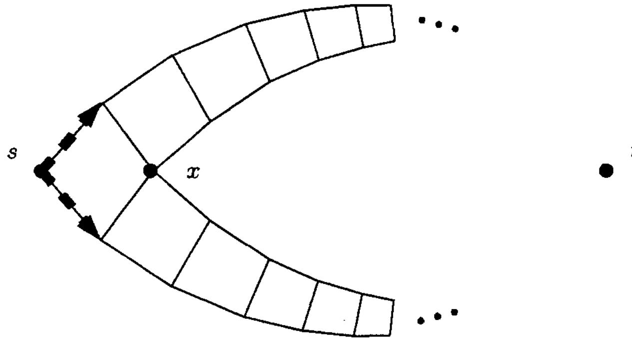


Figure 4.2: Finding two disjoint $s, t$-walks is challenging on a planar graph with non-convex faces. In this graph, only the outer face is non-convex. It is not clear which walk should use $x$.

## 4.2 Agents and scouts

We can think of $s$ transmitting several *agents* responsible for carrying their respective messages to $t$. The agents are independent of each other, and we disallow any direct or indirect communication between agents. Therefore an agent cannot, for example, ask a neighbouring node whether it is currently hosting another agent.

On the other hand, finding $k$ disjoint routes is non-trivial for $k > 2$, so we must provide the agents with some means of computing the local geometry of $G$ outside the immediate neighbourhood of their current nodes. Thus we allow each agent to create a *scout* that can:

- manœuvre through $G$,
- create another scout,
- communicate with its parent (agent or scout), and
- be deactivated (*i.e.*, removed from existence) by its parent.

As with the agents, we restrict the scouts to $O(1)$ memory and local geographic information. Thus, agents and scouts differ primarily in that the former deliver (possibly very large) messages, while the latter are merely delivering topological information to their parents without any additional information.

Consequently, although the agents are required to travel along disjoint walks, we

do not impose such a condition upon the scouts. That is, a scout may visit any vertex regardless of whether it is currently hosting another agent or scout, or has done so in the past. We assume that a scout can recognise and communicate with only its parents, so the scouts cannot pass along information between agents, for instance. These assumption are consistent with the motivations discussed above for the disjoint routing problem: the interception of a scout does not compromise the security of the main data, and the scouts carry so little information relative to the agents that we expect their transmissions to be much less error-prone and to incur little cost.

To find three disjoint $s, t$-walks, we create three agents at $s$, each of which will run the DOUBLE-CROSSING algorithm described below. According to the algorithm, an agent can be in any one of three modes at any time; initially each agent is given a different mode.

An agent running the DOUBLE-CROSSING algorithm sends out scouts in every iteration. The algorithm guarantees the following properties:

- no agent or scout has more than one child scout in existence at any time,
- a scout is only deactivated after its children have been deactivated,
- a third-generation scout never creates children, and
- a scout is restricted to faces incident to its starting point.

In other words, the "family tree" of an agent at any given moment consists of a path of length at most 3. It follows that in total there are never more than nine scouts in existence at any time, which keeps the additional network traffic to a minimum.

Furthermore, observe that the $k$th scout is never farther than $k$ faces away from its ancestral agent. Consequently, if every vertex can precompute and store its local geography up to 3 faces away, we can eliminate scouts and perform all calculations on the precomputed subgraphs. Nevertheless, we will continue to assume that vertices are aware only of their immediate neighbours, and describe the algorithm using scouts.

## 4.3 The DOUBLE-CROSSING algorithm on a sphere

We will present a disjoint routing algorithm for convex polyhedra in $\mathbb{R}^3$ embedded on a sphere, and later show that the same algorithm can be slightly modified for 3-connected convex embeddings in the plane. A well-known result by Steinitz [SR34] shows that convex polyhedra in $\mathbb{R}^3$ and 3-connected planar graphs are essentially the same objects, and Tutte's Theorem [Tut63], also well-known, shows that any 3-connected planar graph has a convex embedding. A convex polyhedron on a sphere is not a realistic model for most routing applications, but it provides a symmetry that simplifies the arguments in our proof of correctness; in the latter geometry the analagous arguments would involve tedious special cases concerning the outer face $f_O$.

Consider a convex polyhedron $P$ embedded on a sphere, keeping the assumptions that vertices are aware of their coördinates (now in $\mathbb{R}^3$) and those of their neighbours. Any vertex $v$ can determine the faces incident to it as follows. Choose any three distinct neighbours; these neighbours along with $v$ itself uniquely determine the sphere $S$ through $P$. Now project all neighbours onto the plane through $v$ tangent to $S$. Radially adjacent neighbours in the projection are on the same face in $P$; to traverse a face in $P$ it is enough to store the plane through $v$ and the two neighbours determining the face.

Let $s$ and $t$ be vertices on $P$. If $s$ and $t$ are adjacent, then it is trivial to find three disjoint $s,t$-paths: send one agent directly from $s$ to $t$, and the other two along the faces incident to $st$. We therefore assume that $s$ and $t$ are distinct and non-adjacent.

Choose a circle $\ell$ through $st$ (say, the shortest circle); analogously to Chapter 2, we will assume for simplicity that (the projections of) no three vertices on $P$ lie on $\ell$. The circle $\ell$ is naturally partitioned into two arcs $st$ and $\overline{st}$. See Figure 4.3. For simplicity of the figures, we will usually draw $P$ as if it were embedded in the plane, *i.e.*, without the circumscribing sphere and curvature of the faces, and depicting $\ell$ as a straight line.

We denote by $\prec$ the natural orderings on $st$ and $\overline{st}$ with respect to distance from $s$. That is, for points $p$ and $q$ on $st$, we write $p \prec q$ if $p$ is closer to $s$ than $q$, and similarly for points on $\overline{st}$. We will never use $\prec$ to compare a point on $st$ to a point on $\overline{st}$.

We now proceed with a detailed description of the DOUBLE-CROSSING algorithm. Send two agents, DOUBLEUP and DOUBLEDOWN (collectively known as the *double agents*), along $st$-crossing faces towards $t$, and send the third agent, SOLITARY (or the *solitary agent*), along $\overline{st}$-crossing faces in the opposite direction; see Figure 4.4.

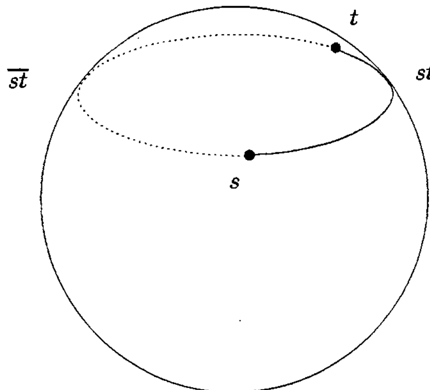


Figure 4.3: The circle $\ell$ through $s$ and $t$ is naturally partitioned into two arcs $st$ and $\overline{st}$. Faces are omitted for clarity.

More precisely, each agent maintains a variable *mode*, which at any time is one of DOUBLEUP, DOUBLEDOWN, or SOLITARY, and may change during the course of the algorithm. Note that we will often identify an agent with its current mode. The three agents use identical routing algorithms, but initially each agent has a different mode. Another variable, *doubleDirection*, determines whether a double agent progresses along $st$- or $\overline{st}$-crossing faces; the solitary agent uses the same variable but travels in the opposite direction (*i.e.*, when $doubleDirection = st$, it travels along $\overline{st}$, and *vice versa*). The meaning of wording such as "travelling along $st$" and "reversing direction" should be clear. Furthermore, for convenience, we will consider *doubleDirection* to be equivalent to either $st$ or $\overline{st}$, so that we may, for example, talk about a face intersecting *doubleDirection*. We also take $\overline{doubleDirection}$ to mean $\ell \setminus \{doubleDirection \setminus \{s,t\}\}$, which has an obvious meaning. In practice, note that *doubleDirection* can be implemented using only one bit.

As with standard face routing, an agent stores the vertex visited immediately prior to the current vertex; this vertex along with the agent's direction determines the current face. For clarity we will henceforth use *currentFace* as a variable, keeping in mind that in practice it is stored as a single vertex.

The initialisation of the three agents is summarised as INIT (Algorithm 7). The source vertex *creates* each agent with several parameters, then *activates* the agent; that is, the agent then becomes independent and follows the routing algorithm DOUBLE-CROSSING (Algorithm 8) until reaching $t$.

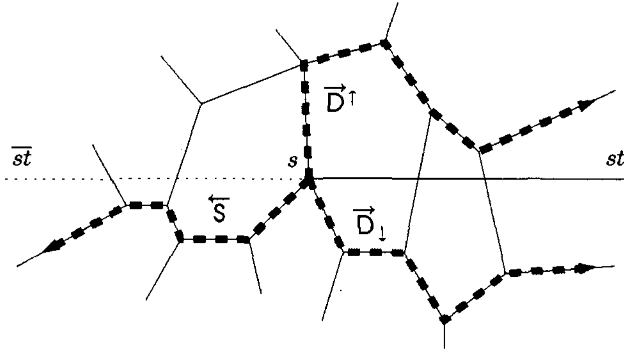


Figure 4.4: The basic approach of the DOUBLE-CROSSING algorithm: send two agents (DOUBLEUP and DOUBLEDOWN) along $st$-crossing faces, and a third agent (SOLITARY) along $\overline{st}$-crossing faces. The agents DOUBLEUP, DOUBLEDOWN, and SOLITARY are indicated by $D^{\uparrow}, D_{\downarrow}$, and S, respectively, in the diagram. Arrows over the symbols indicate direction of travel.

**Algorithm 7** Initialisation

1: **procedure** INIT
2: Let $f$ and $g$ be the faces incident to $s$ intersecting $st$ and $\overline{st}$, respectively.
3: **Create** an agent at $s$ with the following parameters.
4: $mode \leftarrow$ DOUBLEUP
5: $doubleDirection \leftarrow st$
6: $currentFace \leftarrow f$
7: $activeInterval \leftarrow \overline{st}$
8: **Activate** the agent

9: **Create** an agent at $s$ with the following parameters.
10: $mode \leftarrow$ DOUBLEDOWN
11: $doubleDirection \leftarrow st$
12: $currentFace \leftarrow f$
13: $activeInterval \leftarrow \overline{st}$
14: **Activate** the agent

15: **Create** an agent at $s$ with the following parameters.
16: $mode \leftarrow$ SOLITARY
17: $doubleDirection \leftarrow st$
18: $currentFace \leftarrow g$
19: $activeInterval \leftarrow \overline{st}$
20: **Activate** the agent
21: **end procedure**

**Algorithm 8** DOUBLE-CROSSING: main loop

```
1: procedure DOUBLE-CROSSING                                  ▷ v is the current vertex
2:     if v = t then
3:         Terminate
4:     else
5:         Send out a scout to determine whether v is on an active DCC.
6:         if v is on an active DCC (f, g1, g2) then
7:             Call DCC-ADJUST
8:         else
9:             Travel to the next vertex along currentFace according to doubleDirection,
                   (SOLITARY according to overline(doubleDirection)) updating currentFace when
                   appropriate.
                   In DOUBLEUP mode, avoid crossing to a vertex below ℓ.
                   In DOUBLEDOWN mode, avoid crossing to a vertex above ℓ.
                   In SOLITARY mode, avoid travelling towards the central face of an
                   SCC.
10:        end if
11:    end if
12: end procedure
```

If an agent does not encounter any obstacles that we will call *single-crossing configurations* and *double-crossing configurations*, or *SCCs* and *DCCs*, respectively, then it will successfully reach $t$ by simple face traversal. On the other hand, if an agent detects one of these configurations, it will adjust in several ways, to be described shortly, to avoid colliding with the walks taken by the other agents.

In particular, the adjustments for SCCs are simple, whereas DCCs require some careful manœuvring. Double-crossing configurations are the main objects of interest in this problem, hence the name of the algorithm.

The adjustments at DCCs are chosen in such a way that after the agents emerge from the configuration, there are again two double agents travelling towards $t$ from one direction and one solitary agent travelling towards $t$ from the opposite direction. Note that in practice the agents do not necessarily reach a DCC at the same time. However, in the proof, we will manipulate their timing to synchronise them at DCCs.

We now define double-crossing configurations and give a procedure describing the adjustments for them; a similar discussion for SCCs will follow.

Let $f$, $g_1$, and $g_2$ be an $\overline{st}$-crossing face and two $st$-crossing faces, respectively, with the following properties. Assume $f$ is incident to $g_1$ and $g_2$ such that $f \cap g_1$ is above

$\ell$ and $f \cap g_2$ is below $\ell$ (note that if the intersection of an $st$-crossing face and an $\overline{st}$-crossing face is an edge, both endpoints must be above or below $\ell$). Also assume that $f$ is not incident to any $st$-crossing face $g'$ such that $g' \cap st \prec g_1 \cap st$ or $g' \cap st \prec g_2 \cap st$, and similarly that for $i \in \{1,2\}$, $g_i$ is not incident to any $\overline{st}$-crossing face $f'$ such that $f' \cap \overline{st} \prec f \cap \overline{st}$. That is, $f$, $g_1$, and $g_2$ are $\prec$-minimal with respect to intersections with $\ell$.

We say that the triple $(f, g_1, g_2)$ constitutes a *double-crossing configuration*. If $g_1 \cap st \prec g_2 \cap st$, then we say that $f$, $g_1$ and $g_2$ are the *central*, *near* and *far* faces in the double-crossing configuration, respectively, and that the configuration is an *upper DCC*. It will also be convenient to refer to $g_1$ and $g_2$ as the *upper* and *lower* faces in the configuration, respectively. Similarly, if $g_2 \cap st \prec g_1 \cap st$, then $f$, $g_2$ and $g_1$ are the central, near (lower) and far (upper) faces, respectively, in a *lower DCC*. Thus, a face is near or far according to its distance from $s$ with respect to the ordering $\prec$, and a DCC is upper or lower according to its near face. We similarly define another set of DCCs by transposing all occurrences of $st$ and $\overline{st}$ in the above definition. Figure 4.5 shows a DCC on the sphere.

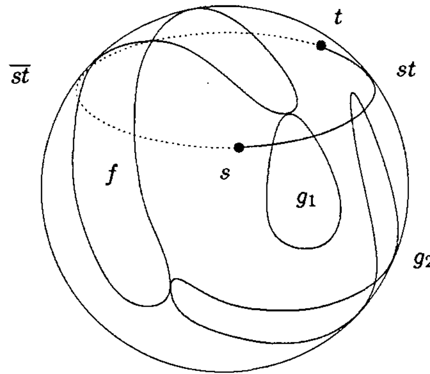


Figure 4.5: A double-crossing configuration $(f, g_1, g_2)$. The configuration is an upper DCC: $f$ is the central face, $g_1$ is the upper (and near) face, and $g_2$ is the lower (and far) face. The faces are drawn as curves on the sphere for clarity; we may take the curves to represent the projections of the actual faces.

Naïve face-routing will fail on DCCs. Figure 4.6 demonstrates why the configurations are problematic: if the double agents make no adjustments at $g_1$ and $g_2$, then the solitary agent can traverse neither the upper nor the lower half of $f$ without colliding with another agent's walk.

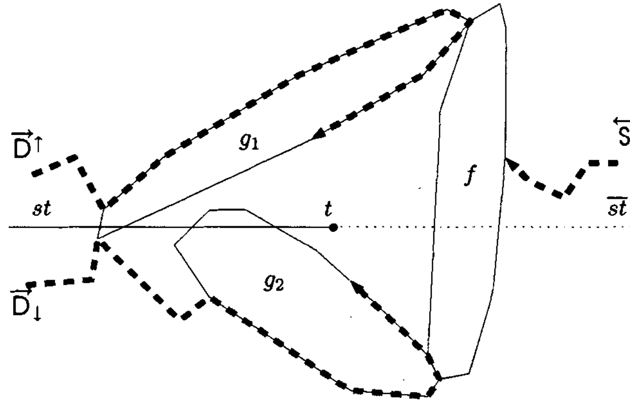


Figure 4.6: If agents do not adjust to DCCs, collisions are inevitable. Here some vertex on $f$ will be used by two agents: the solitary agent must traverse either the top or bottom half of $f$, but the double agents will also collectively visit both sides of the face.

The agents avoid the situation in Figure 4.6 by adjusting as described in DCC-ADJUST (Algorithm 9). In short, the adjustments are chosen so that two double agents emerge from the DCC from one direction, and a solitary agent emerges from the other direction, as we stated earlier. Figure 4.7 illustrates the general case. Note that after the adjustments, the (new) DOUBLEUP agent may be below $\ell$ on the central face $f$, but this is fine: in the following iterations it will naturally travel around $f$ until reaching a vertex above $\ell$, and thenceforth continue as usual. This is consistent with the behaviour specified in Algorithm 8: namely, we require that DOUBLEUP avoid crossing to a vertex below $\ell$, as opposed to requiring that it avoid choosing any $\ell$-crossing edge. The same comment applies similarly to the new DOUBLEDOWN agent.

However, the agents need not adjust to every DCC. Figure 4.8 shows the same configuration as in Figure 4.7, but with the agents approaching from the opposite direction. Note that the double agents treat $g_1$ and $g_2$ as any other $\ell$-crossing faces, and that the solitary agent makes only minor adjustments (these will be explained when we discuss single-crossing configurations, below). It should be clear that the decision to adjust to a DCC depends on the agent's direction and mode, and the relative position of the faces composing the configuration.

The decision to adjust involves one more parameter, the *active interval*: each agent stores a subset $activeInterval \subseteq \ell$ within which it searches for DCCs – more precisely, the agent considers a DCC only if the faces composing the configuration intersect $\ell$

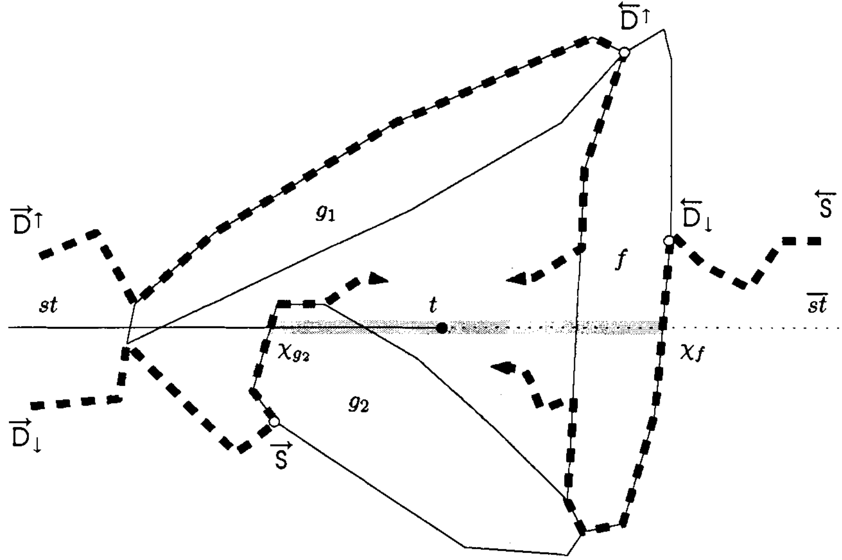


Figure 4.7: The correct adjustments at a double-crossing configuration $C$. Several iterations are shown after the adjustments to illustrate the fundamental behaviour of the agents: after adjusting to $C$, there are two double agents approaching $t$ from one direction, and a solitary agent approaching from the other. The shaded region $\chi_{g_2}t$ along $\ell$ shows the new active interval (for all three agents) after the adjustments. Note that the new DOUBLEDOWN agent is initially at a vertex above $\ell$.

within *activeInterval*. Initially, $activeInterval = \ell$, and the interval diminishes each time the agent adjusts to a DCC. When the agent arrives at $t$ we can assume for convenience that $activeInterval = \{t\}$. The details of the update are discussed shortly.

A DCC $(f, g_1, g_2)$ is said to be ***active*** (with respect to an agent and its current parameters) if the following conditions hold: $f$, $g_1$, and $g_2$ intersect $\ell$ within *activeInterval*, and if they are $\prec$-minimal with respect to these intersections in the sense that no

- $f \cap \overline{doubleDirection} \subset activeInterval$,
- $g_1 \cap doubleDirection \subset activeInterval$,
- $g_2 \cap doubleDirection \subset activeInterval$, and
- $f$, $g_1$, and $g_2$ are $\prec$-minimal in the sense that there exists no $\overline{doubleDirection}$-crossing face $f'$ such that $f' \cap \ell \prec f \cap \ell$, $f' \cap doubleDirection \subset activeInterval$, and $(f', g_1, g_2)$ constitutes a DCC, and similarly for $g_1$ and $g_2$.

**Algorithm 9** DCC adjustment

```
1: procedure DCC-ADJUST    ▷ The current vertex v is on an active DCC (f, g1, g2).
2:     Let χf, χg1, and χg2 be as described in the text.
3:     if mode = DOUBLEUP then                                      ▷ v is on g1
4:         if g1 is the near face of the DCC then
5:             Traverse the upper half of g1 until reaching a vertex on f.
6:             doubleDirection ← ¬doubleDirection
7:             currentFace ← f
8:             activeInterval ← χg2 χf
9:         else g1 is far
10:            mode ← SOLITARY
11:            currentFace ← g1
12:            activeInterval ← χg1 χf
13:        end if
14:    else if mode = DOUBLEDOWN then                               ▷ v is on g2
15:        if g2 is the near face of the DCC then
16:            Traverse the lower half of g2 until reaching a vertex on f.
17:            doubleDirection ← ¬doubleDirection
18:            currentFace ← f
19:            activeInterval ← χg1 χf
20:        else g2 is far
21:            mode ← SOLITARY
22:            currentFace ← g2
23:            activeInterval ← χg2 χf
24:        end if
25:    else mode = SOLITARY                                         ▷ v is on f
26:        if g1 is the near face of the DCC then
27:            mode ← DOUBLEDOWN
28:            currentFace ← f
29:            activeInterval ← χg2 χf
30:        else g2 is near
31:            mode ← DOUBLEUP
32:            currentFace ← f
33:            activeInterval ← χg1 χf
34:        end if
35:    end if
36: end procedure
```

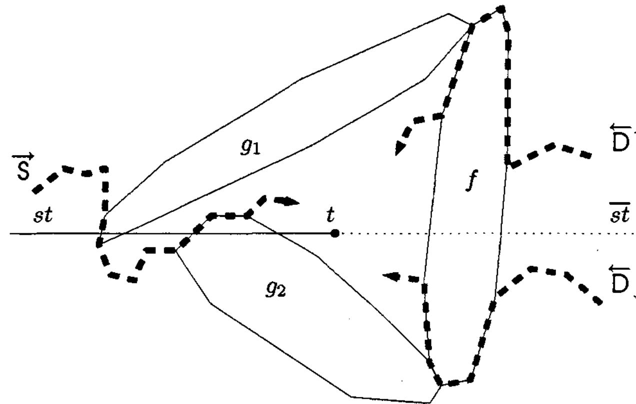


Figure 4.8: A DCC $(f, g_1, g_2)$, approached from the "safe" direction – the DCC is not active for any of the agents. The double agents make no adjustments, while the solitary agent adjusts to the SCCs $(g_1, f)$ and $(g_2, f)$.

We now explain and justify the rôle of the *activeInterval* parameter in DCC-ADJUST: note that every adjustment concludes by redefining the interval. Let $h$ be an $\ell$-crossing face. Then define $\chi_h$ to be the intersection of $h$ with $\ell$, choosing the intersection to be the farther one from $t$ with respect to $\prec$. The new active interval is $\chi_g\chi_f$ where $g$ and $f$ are the far and central faces of the DCC, respectively; that is, it consists of the longest segment of $\ell$ between the far and central faces, as shown in Figure 4.7, above.

The points of intersection are chosen to be the farther one from $t$ to account for the possibility that the far face of one DCC is the central face of the next, as illustrated in Figure 4.9. Notice that two of the agents adjust to $(g_2, f_2, f_1)$ immediately after adjusting to $(f_1, g_1, g_2)$. This is indicated in the figure as $\mathsf{M}_1/\mathsf{M}_2$, where $\mathsf{M}_1$ is the new mode (and direction) after the first adjustment, and $\mathsf{M}_2$ after the second.

It is necessary to use the active interval to handle graphs that contain DCCs symmetrically *entangled* around $t$ as shown in Figure 4.10. The figure illustrates the purpose of the active interval: the agents first arrive at the configuration $C$, and after the adjustments, $C'$ is no longer active. On the other hand, if the agents are less discerning and do not take the active interval into consideration, then (at least) two of the agents will visit the same vertex, as shown in Figure 4.11. In this case, notice that all three agents adjust to $C$, similarly to the previous example. However, only one agent (the one originally in DOUBLEUP mode) adjusts to $C'$, since although it is effectively active for all agents, the other two agents have already bypassed it and hence will not detect

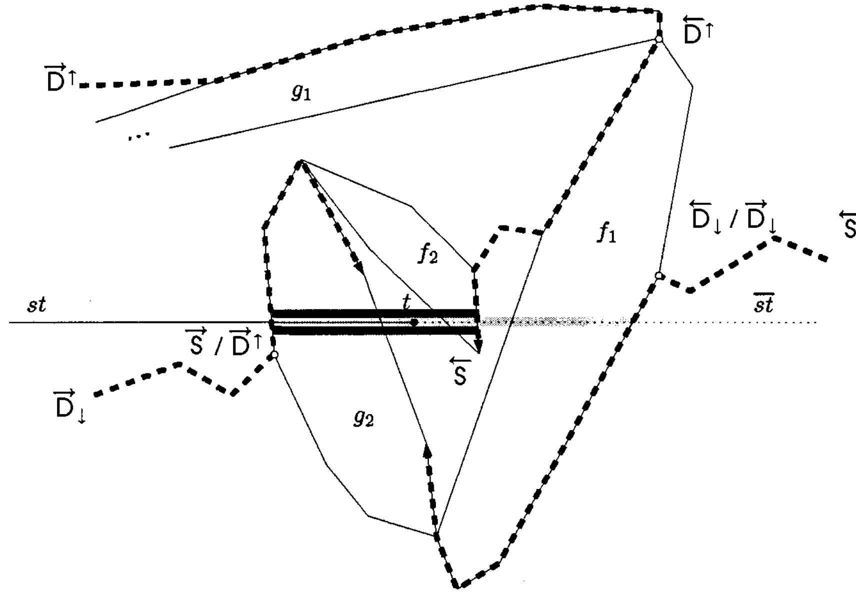


Figure 4.9: The DCCs $C_1 = (f_1, g_1, g_2)$ and $C_2 = (g_2, f_2, f_1)$ have two faces in common. After adjusting to $C_1$, two of the agents immediately adjust to $C_2$, as indicated by the slash notation. The shaded regions along $\ell$ show the active intervals after adjusting to $C_1$ (light grey) and $C_2$ (dark grey). For clarity, the $st$-crossing face $g_1$ is only partially shown.

or adjust to it. The same problem occurs if $f = g_1'$ in Figure 4.11, that is, if $f$ and $f'$ are incident and their point of intersection is above $\ell$.

We now turn our attention to SCCs. Let $f$ be an $\overline{st}$-crossing face incident to an $st$-crossing face $g$ such that $f \cap g$ is above $\ell$, and suppose $f$ is not incident to any $st$-crossing face $g'$ such that $f \cap g'$ is below $\ell$. Then we say that the pair $(f, g)$ constitutes a *single-crossing configuration*. We similarly define another set of SCCs by transposing all occurrences of $st$ and $\overline{st}$ in the above definition.

An SCC could be considered an "incomplete" DCC. Only the solitary agent adjusts at an SCC; it does so simply by avoiding $f \cap g$, as shown in Figure 4.12. DOUBLEDOWN is omitted from the diagram to emphasise that the solitary agent is specifically avoiding collision with DOUBLEUP, and that the double agents do not make adjustments.

We observe in passing that, in general, a DCC approached from the "wrong" direction consists of two SCCs with one face in common. This is illustrated above in Figure 4.8.

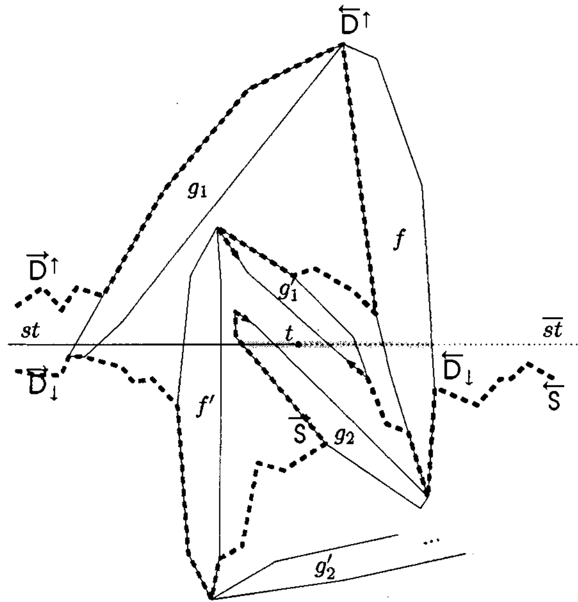


Figure 4.10: The correct behaviour at DCCs $C = (f, g_1, g_2)$ and $C' = (f', g_1', g_2')$ entangled around $t$. The $\overline{st}$-crossing face $g_2'$ is only partially shown. The agents first detect $C$, and after adjusting, $f' \cap \ell$ is no longer in the active interval.

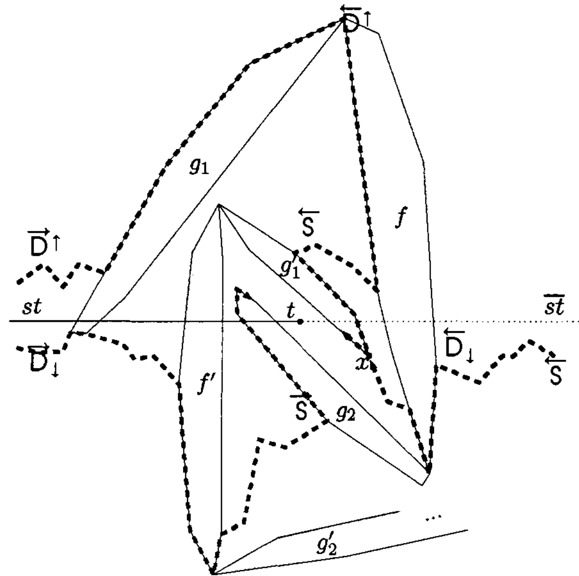


Figure 4.11: If the agents do not maintain an active interval, entangled DCCs will induce collisions; in this example two agents visit $x$. Note that the agent originally in DOUBLEUP mode is the only one to adjust to $C'$, so after all the adjustments as shown, there are two agents in SOLITARY mode, contrary to the design of the algorithm.

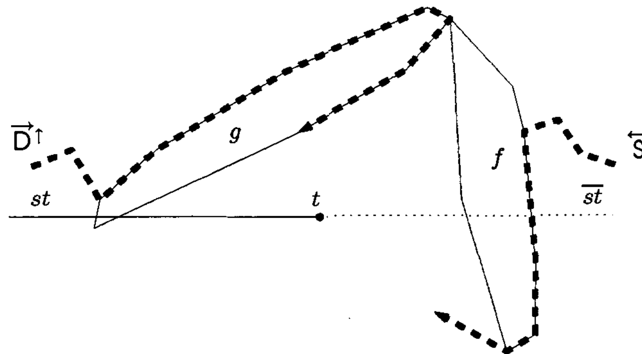


Figure 4.12: A single-crossing configuration $(f, g)$. The solitary agent adjusts by moving to the lower half of $f$ to avoid $g$. The double agents make no adjustments at an SCC.

Let us take a brief look at DCCs and SCCs from another perspective. As we have discussed, the double agents are essentially following the straightforward algorithm for finding two disjoint $s,t$-walks. Now consider the solitary agent. In general, it can choose to traverse its current face either from below $\ell$ or above. For efficiency, we may assume it makes a greedy choice in this respect; for example, see Figure 4.13. The solitary agent's freedom is slightly restricted by an SCC – it no longer has a choice between the upper or lower half of the current face, but nevertheless continues its face routing. The agent's freedom is eliminated entirely by a DCC: all three agents adjust in order to accommodate the solitary agent. In summary, we propose that the DOUBLE-CROSSING algorithm is fundamentally concerned with the local geometry of the solitary agent.

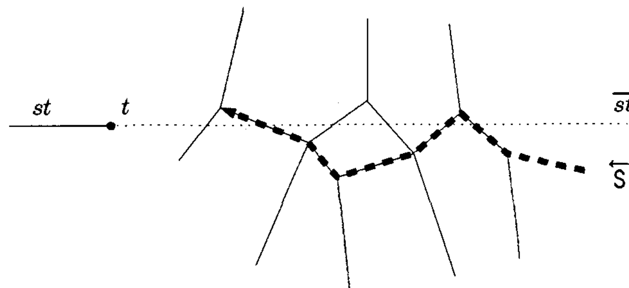


Figure 4.13: In general (*i.e.* when not at an SCC or DCC), the solitary agent can choose to cross $\ell$. Here the choices are greedy with respect to distance to $t$.

The agents detect DCCs and SCCs using scouts – indeed, the possibility of these

configurations is the scouts' *raison d'être*. We briefly present how the detection is managed by the agents and scouts; we will not provide a formal algorithm for each scout, but rather implicitly assume that an agent on a DCC or SCC can determine topological information about that configuration.

Consider an agent in DOUBLEUP mode; without loss of generality, assume it is travelling along $st$. At every iteration, the agent creates a scout to traverse every face incident to the current vertex. Every such face $g$ that crosses $st \cap activeInterval$ is potentially a near or far face in a DCC; the scout therefore traverses $g$ (again) and at each vertex $u$ on $g$ above $st$ creates another scout to traverse faces incident to $u$. The first scout waits at $u$ until its child reports to it before proceeding around its current face.

If the second scout finds an $\overline{st}$-crossing face $f$ with $f \cap \overline{st} \subset activeInterval$, it similarly traverses $f$ and dispatches a third scout at every vertex below $st$ to look for another $st$-crossing face $h$ incident to $f$, such that $h \cap st \subset activeInterval$. Again, the second scout waits for its child to return before proceeding.

If the third scout does find such a face $h$, it returns to its parent and reports success, along with $O(1)$ topological information. The second scout can then report success to its parent, and so on. An agent in DOUBLEDOWN mode uses scouts in an analogous manner; and an agent in SOLITARY mode sends a scout to first find a potential central face of a DCC, and the second and third scouts check for near and far faces.

Finally, observe that the scouts naturally first find $\prec$-minimal faces as described in the definition of active DCC, so an agent is clearly on an active DCC if and only if the scouts detect the DCC.

**Theorem 4.1** *Let $P$ be a convex polyhedron embedded on a sphere, and let $s, t \in V(P)$ be distinct, non-adjacent vertices. Suppose $s$ creates three agents with destination $t$ as described in* INIT. *Then all three agents successfully route to $t$ using the* DOUBLE-CROSSING *algorithm, and the walks described by the agents are mutually internally (vertex-) disjoint.*

*Proof.* When we refer to a DCC in this proof we will assume it is active unless otherwise noted. We will use induction on the number $k$ of DCCs encountered by the agents thus far to show that the agents progress towards $t$ and traverse mutually internally disjoint walks. We first show that the agents encounter the DCCs in the same order,

thus determining a sequence $C_1, C_2, \ldots, C_k$ of DCCs; we may think of the agents as moving from one configuration to the next in phases. This does not affect the behaviour of the agents, since they rely only on geometric information and do not communicate with each other. The coördination of the agents will ensure that at any time in our proof, there is one agent in DOUBLEUP mode, one in DOUBLEDOWN mode, and one in SOLITARY mode.

It is easy to see that an agent must reach either $t$ or a DCC after a finite number of iterations: the agent travels along faces progressively closer to $t$, and reverses direction only at a DCC. We will say that the agents' respective iterations of the main loop (Algorithm 8) occurring between the $j$th and $j+1$st DCCs (or between the $j$th DCC and arrival at $t$) collectively constitute the $j$th *phase* of the algorithm.

Specifically, the 0th phase begins after all agents have been created by INIT (Algorithm 7); for $0 < j \leq k$ the $j$th phase begins at the iteration in which the agents detect the $j$th DCC. For $0 \leq j \leq k$ the $j$th phase ends immediately prior to the iteration in which the agents detect another DCC or are at $t$. Note that since agents send out scouts for DCC detection at the beginning of each iteration, the last vertex visited by an agent in the $j$th phase is the agent's first vertex of the $j+1$st phase. We will say that an agent *arrives* at a DCC $C$ at the end of the $j$th phase if it detects $C$ at the beginning of the $j+1$st phase. An agent may use vertices on inactive DCCs during a phase, but we do not consider the agent to have arrived at these DCCs.

Observe that after any adjustments in DCC-ADJUST (Algorithm 9), an agent's mode and direction are constant for the remainder of each phase; when we specify an agent by its mode and direction during a phase we therefore understand the parameters to refer to those after the adjustments.

Since the agents arrive at the same DCCS, they agree on a common active interval $I_j$ at the $j$th phase. The active interval decreases at each phase, *i.e.*, $\ell =: I_0 \supset I_1 \supset \ldots \supset I_k$, but we will prove a stronger result. At the $j$th phase, $0 \leq j \leq k$, we define an *active region* $R_j$ that extends the notion of the active interval. We will show that the agents remain within $R_j$ during that phase, and that $R_j$ is contained in $R_{j-1}$ for every $j > 0$. Also, $I_j$ is naturally contained in $R_j$.

Define $R_0$ to be the polyhedron $P$. Now let $0 < j \leq k$ and suppose the $j$th DCC detected by the agents is $C = (f, g_1, g_2)$. Then define $R_j$ to be the union of the region above $\ell$ bounded by $f$ and $g_1$, and the region below $\ell$ bounded by $f$ and $g_2$. Thus, $R_j$ is

the region bounded by $\ell$ and the faces composing $C$, except for the part of the near face extending beyond $\ell$, as illustrated in Figure 4.14. Evidently $R_j$ contains $I_j$. In the final phase we can consider the active region to consist of the vertex $t$.

Again, since the agents detect DCCs in the same order, the $R_j$ are common to all agents. It is important to note that the active regions are used only in the proof; the agents do not store or calculate any $R_j$.

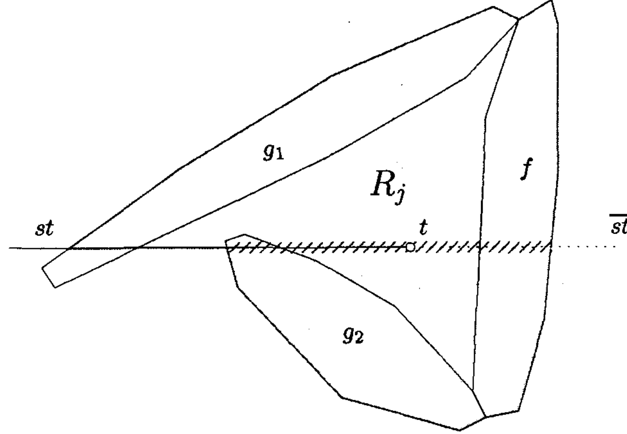


Figure 4.14: The active region $R_j$. Note that $R_j$ does not include all of the near face $g_1$. The active interval $I_j$ is shown as a hatched strip.

As with the main theorems in Chapter 2, for the remainder of this proof we will use induction on a number of interdependent claims.

**Claim 4.1** *Let $k \geq 0$. During the kth phase, there is one agent in* DOUBLEUP *mode, one in* DOUBLEDOWN *mode, and one in* SOLITARY *mode. If k is even (resp. odd), the double agents travel along $st$ (resp. $\overline{st}$) and the solitary agent travels along $\overline{st}$ (resp. $st$). Also, each agent has the same active interval $I_k$ and active region $R_k$ during the kth phase.*

*Moreover, if $k > 0$, then at the beginning of the kth phase either all three agents are at $t$, or they are all on a common DCC $C_k = (f, g_1, g_2)$. In the latter case, then if $C_k$ is an upper (resp. lower) DCC, during the adjustments* DOUBLEUP *(resp.* DOUBLEDOWN*) moves along at least one edge during the adjustments, while the other two agents remain in place. After the adjustments,* DOUBLEUP *is on $f \cap g_1$ (resp. $f \setminus \{g_1, g_2\}$),* DOUBLE-DOWN *is on $f \setminus \{g_1, g_2\}$ (resp. $f \cap g_2$), and* SOLITARY *is on $g_2 \setminus f$ (resp. $g_1 \setminus f$), and all three agents are on the boundary of $R_k$.*

*Proof.* The specifications in the INIT procedure and the definition $R_0 := P$ guarantee that the claim holds when $k = 0$. Now let $k > 0$. By induction on Claim 4.2, below, the agents arrived at a common DCC $C_k = (f, g_1, g_2)$ at the end of the $k-1$st phase, with DOUBLEUP, DOUBLEDOWN, and SOLITARY on $g_1 \setminus f$, $g_2 \setminus f$, and $f \setminus \{g_1, g_2\}$, respectively, all on the boundary of $R_k$. We immediately have that $I_k$ and $R_k$ are well-defined, and they obviously do not change during the $k$th phase.

At the beginning of the $k$th phase, the agents adjust according to DCC-ADJUST. Without loss of generality, assume that $C_k$ is an upper DCC. Then the former DOUBLEUP agent (*i.e.*, the agent in DOUBLEUP mode in the $k-1$st phase) moves (using at least one edge) to $f$ and reverses direction; the former DOUBLEDOWN agent switches to SOLITARY mode; and the former SOLITARY agent switches to DOUBLEDOWN mode. Neither the former DOUBLEDOWN nor SOLITARY agent move during the adjustments, so obviously the new SOLITARY and DOUBLEDOWN agents will be on $g_2 \setminus f$ and $f \setminus \{g_1, g_2\}$, respectively. All three agents remain on the boundary of $R_k$ during the adjustments.

Clearly the three agents are in distinct modes after the adjustments, the double agents' (common) direction is the reverse of the former double agents' direction, and the solitary agent's direction is similarly the reverse of the former solitary agent's direction. This proves the claim. □

**Claim 4.2** *At the end of the kth phase, either all three agents arrive at t, or they arrive at a common DCC $C_{k+1} = (f, g_1, g_2)$. In the latter case,* DOUBLEUP, DOUBLEDOWN, *and* SOLITARY *arrive at vertices on $g_1 \setminus f$, $g_2 \setminus f$, and $f \setminus \{g_1, g_2\}$, respectively, on the boundary of $R_{k+1}$.*

*Proof.* We start by considering the double agents during the $k$th phase. By Claim 4.1, the double agents travel in the same direction during the $k$th phase; assume without loss of generality that their direction is $st$. From the same claim we also have that the active interval $I_k$ is common to all agents. For the remainder of this argument we will denote by $v^\uparrow$ and $v_\downarrow$ the locations of DOUBLEUP and DOUBLEDOWN, respectively, after the adjustments at the beginning of the $k$th phase.

Towards a contradiction, suppose that the double agents do not arrive at the same DCC at the end of the phase. There are two possibilities: either one of the agents arrives at $t$ and the other at a DCC, or the agents arrive at different DCCs.

In the first case, again without loss of generality, assume that DOUBLEUP arrives at $t$ at the end of the $k$th phase, but that DOUBLEDOWN arrives at a double-crossing configuration $C = (f, g_1, g_2)$. It follows that $f \cap \overline{st} \subset I_k$. Let $U$ be the region above $\ell$ containing $t$ and bounded by $f$, $g_1$, and $\ell$; see Figure 4.15. If $v^{\uparrow}$ is outside $U$, then by the polygonal version of the Jordan Curve Theorem, and DOUBLEUP's choice of faces, any $v^{\uparrow}, t$-walk must use a vertex on $g_1$. Now since DOUBLEUP arrives at $t$, and not $C$, at the end of the $k$th phase, $C$ must be inactive for the agent during this phase. There are two ways this can happen: either DOUBLEUP reverses direction before using a vertex on $C$, or $f \cap \overline{st} \not\subset I_k$. But an agent's direction reverses only after arriving at a DCC, and we have already seen that $f \cap \overline{st} \subset I_k$. It follows that $v^{\uparrow}$ is inside $U$, and that $v^{\uparrow}$ is neither on $g_1$ nor on a vertex of $f$ on the boundary of $U$. Consequently, $k > 0$, for $s$ cannot be inside $U$ unless it is a vertex of $f$ or $g_1$ on the boundary of $U$. Therefore, by Claim 4.1, $v^{\uparrow}$ and $v_{\downarrow}$ both lie on the central face $f' \neq g_1$ of some DCC $C'$. But then $I_k$ extends only as far to the left as $f' \cap st$, so $g_1$ does not intersect $st$ within the active interval, contradicting the assumption that DOUBLEDOWN arrives at $C$.

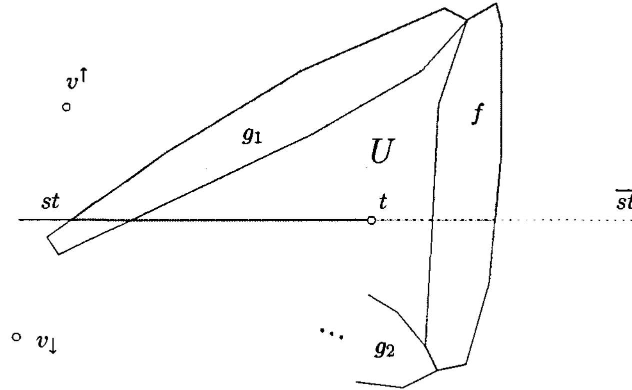


Figure 4.15: DOUBLEUP cannot find a $v^{\uparrow}, t$-walk without passing through $g_1$.

Let us now examine the second case: suppose that DOUBLEUP and DOUBLEDOWN arrive at distinct DCCs $C = (f, g_1, g_2)$ and $C' = (f', g_1', g_2')$ at the end of the $k$th phase. Assume without loss of generality that $f \cap \overline{st} \prec f' \cap \overline{st}$. Then clearly $g_1 \cap st \prec g_1' \cap st$ and $g_2 \cap st \prec g_2' \cap st$. Since one of the agents detects $C$, it follows that $f \cap \overline{st} \subset I_k$. But, similarly to the reasoning above, we can appeal to the polygonal Jordan Curve Theorem and the agents' choice of faces to show that both agents must arrive at $C$ at the end of the $k$th phase, a contradiction.

So far we have shown that the double agents in the $k$th phase must arrive at the same DCC at the end of that phase. It remains to show that the solitary agent also arrives at the same DCC. Again towards a contradiction, suppose not. There are three cases, which we will not discuss in detail since the arguments are similar to those above. Either the double agents arrive at $t$ and the solitary agent arrives at a DCC; the double agents arrive at a DCC and the solitary agent arrives at $t$; or the double agents and the solitary agent arrive at distinct DCCs. The first two cases are dismissed with the polygonal Jordan Curve Theorem; the argument for the third case is similar to that showing that the double agents cannot arrive at distinct DCCs.

Therefore all three agents arrive at $t$ or at the same DCC $C_{k+1} = (f, g_1, g_2)$ at the end of the $k$th phase, as required. Consequently, $R_{k+1}$ is well-defined; it is easy to see that the agents arrive on the boundary of $R_{k+1}$. It is clear from the definition of active DCCs that DOUBLEUP, DOUBLEDOWN, and SOLITARY arrive at the upper, lower, and central faces of $C_{k+1}$, respectively. We must now refine this result to show that DOUBLEUP, DOUBLEDOWN, and SOLITARY arrive at vertices on $g_1 \setminus f$, $g_2 \setminus f$, and $f \setminus \{g_1, g_2\}$, respectively.

Suppose that DOUBLEUP arrives at a vertex $w$ on $g_1 \cap f$ at the end of the $k$th phase. If the agent's walk to $w$ during that phase has length at least 1, let $w'$ be the penultimate vertex the walk. Clearly $w'$ cannot also be on $g_1$ for then DOUBLEUP would have arrived at $w'$, not $w$. Therefore, $w'$ is on some other face $\tilde{g}$ for which $\tilde{g} \cap \ell \prec g_1 \cap \ell$. Evidently $\tilde{g}$ is the upper face in a DCC $\tilde{C}$ $(f, \tilde{g}, g_2)$; it follows that $\tilde{C}$ is inactive since the agent does not arrive on it at $w'$. Therefore the active interval $I_k$ does not extend as far left as $\tilde{g} \cap st$; it extends as far as $f' \cap st$, where $f'$ is the central face of $C_k$; see Figure 4.16(b). However, by Claim 4.1, DOUBLEUP is on $f'$ at the beginning of the $k$th phase. It therefore cannot reach $w$ through $w'$ since DOUBLEUP travels along faces progressively closer to $t$, contradicting the assumption that $w'$ was the penultimate vertex in the walk.

We must therefore consider the possibility that DOUBLEUP's walk during the $k$th phase had length 0, *i.e.*, that it was already at $w$ at the beginning of the $k$th phase. By Claim 4.1, $C_k$ cannot be an upper DCC, for otherwise DOUBLEUP would have moved along at least one edge during the adjustments. Thus we have that $C_k$ is a lower DCC, and that $w$ is not on the upper face $g_1'$ of $C_k$. But since $C_{k+1}$ is active, $g_1' \cap \overline{st} \prec f \cap \overline{st}$, implying that $w$ is not on the boundary of $R_k$, as shown in Figure 4.16(b). This

contradicts Claim 4.1.

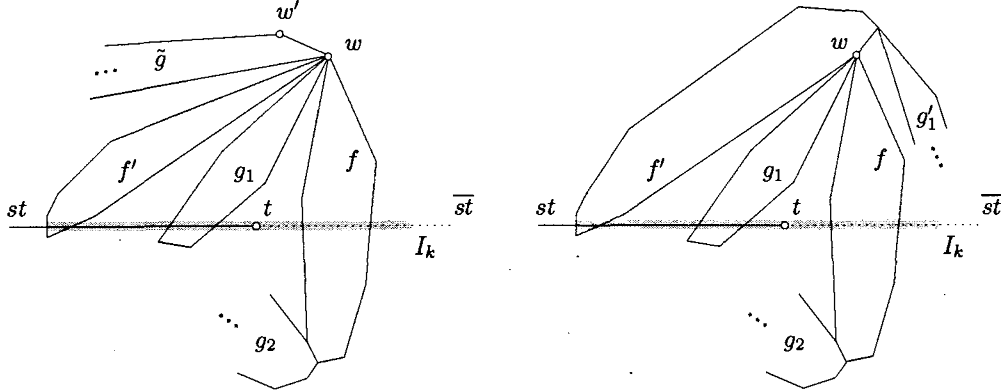


Figure 4.16: DOUBLEUP cannot arrive at a vertex $w$ on $g_1 \cap f$: (a) the case where it travels along at least one edge in the $k$th phase; (b) the case where it remains in place during the $k$th phase.

The arguments for DOUBLEDOWN and SOLITARY are similar. □

**Claim 4.3** *If $k > 0$, then $R_k \subset R_{k-1}$. Thus, $P =: R_0 \supset R_1 \supset \ldots \supset R_k$.*

*Proof.* That $R_k$ is defined for all $k \geq 0$ follows from Claim 4.1 Trivially we have $R_1 \subset R_0 := P$, so assume $k > 1$. If the agents are at $t$ at the beginning of the $k$th phase, we have $R_k := \{t\}$, which is also trivial; therefore assume that the agents are at a DCC at the beginning of the $k$th phase.

Let $f$ and $g$ be the central and far faces of $C_{k-1}$, respectively, and let $C_k = (f', g'_1, g'_2)$. Then $f' \cap \ell \subseteq I_{k-1}$, so $g \cap \ell \preceq f' \cap \ell$ (note that possibly $f' = g$). Similarly, $f \cap \ell \preceq g'_i \cap \ell$ for $i \in \{1, 2\}$; again note that possibly $g'_1 = f$ or $g'_2 = f$, but in particular $f \cap \ell \precneqq g' \cap \ell$, where $g'$ is the far face of $C_k$. Observe that this implies that $I_k \supset I_{k-1}$.

Now suppose that $R_k \not\subseteq R_{k-1}$. Then there exists a point $w \in R_k \setminus R_{k-1}$; in particular, since $R_k$ is a closed set, we may choose $w$ to be a boundary point of $R_k$. By definition of $R_k$, $w$ lies on one of the faces composing $C_k$ or on the interval of $\ell$ between the near and far faces of $C_k$. If $w$ is on $f'$, then since $g \cap \ell \preceq f' \cap \ell$, $f'$ must cross one of the faces composing $C_{k-1}$, a contradiction. Similarly $w$ cannot lie on $g'_1$ or $g'_2$. Thus $w$ is on the interval of $\ell$ between $g'_1 \cap \ell$ and $g'_2 \cap \ell$. But we have already shown that $f \cap \ell \preceq g'_i \cap \ell$ for $i \in \{1, 2\}$, so $w$ is contained in $I_{k-1} \subset R_{k-1}$, another contradiction. Hence there can be no such point $w$, so $R_k \subseteq R_{k-1}$.

Note that although $R_k$ may contain $f$ and $g$, it obviously cannot contain every point on the near face of $C_{k-1}$. Therefore the containment is proper; *i.e.*, $R_k \subset R_{k-1}$. Consequently, $R_0 \supset R_1 \supset \ldots \supset R_k$. □

Claim 4.3 proves that the algorithm must terminate, for the active region is determined by three faces and therefore cannot decrease indefinitely. It remains to prove that the agents describe disjoint walks; we will require one more result before the final step.

**Claim 4.4** *During the kth phase, including the initial adjustments (when $k > 0$), every agent remains within $R_k$.*

*Conversely, when $k > 0$, no agent has used a vertex in $R_k$ until arriving at $C_k$ (or $t$) at the end of the $k-1$st phase.*

*Proof.* The statement is trivial for $k = 0$, so assume $k > 0$. The statement is also trivial if the agents are at $t$, so assume they are at $C_k = (f, g_1, g_2)$ at the beginning of the $k$th phase. It is clear from DCC-ADJUST that the agents remain in $R_k$ during the adjustments. For the remainder of the $k$th phase, DOUBLEUP traverses only those faces intersecting $\ell$ between $t$ and $f$. These faces clearly cannot contain any vertices outside $R_k$ without crossing one of the faces composing $C_k$, so DOUBLEUP remains within $R_k$. Moreover, any vertices on the boundary of $R_k$ visited by the agent must lie on $f$. Similar arguments show that the other two agents also remain within $R_k$; also, any boundary vertices used by DOUBLEDOWN are on $f$, and any boundary vertices used by SOLITARY are on the far face of $C_k$.

Now, towards a contradiction, suppose that an agent uses a vertex $w$ in $R_k$ before the end of the $k-1$st phase; assume $w$ is the first such vertex. Let $j$ be the least number for which the agent uses $w$ before the end of the $j$th phase. We will show that $j$ must be $k-1$. Indeed, if $j < k-1$, then by Claim 4.3 we have $R_k \subset \ldots \subset R_{j+1}$; however, by the induction hypothesis, no agent uses any vertex in $R_{j+1}$ until the end of the $j$th phase. Therefore $j = k-1$.

Suppose that DOUBLEUP is the agent that uses $w$ in the $k$–1st phase before arriving at $C_k = (f, g_1, g_2)$. Then $w$ cannot be on $g_1$, for the agent would have detected $C_k$ at $w$, marking the end of the $k-1$st phase. We can assume that $w$ is on the boundary of $R_k$, since the agent cannot reach the interior of $R_k$ without first passing through $g_1$. Also, $w$ is not on $\ell$ for then it would be collinear with $s$ and $t$. Therefore $w$ is above $\ell$ and

either on $f$ or $g_2$. If $w$ is on $f$, then it also lies on some face $g'$ incident to $f$ for which $g' \cap \ell \prec g_1 \cap \ell$; see Figure 4.17. But using similar arguments as in the latter half of the proof of Claim 4.2 shows that this is impossible. The only remaining possibility is that $w$ lies on $g_2$. But either $C_k$ is an upper DCC, in which case the vertices of $g_2$ above $\ell$ are in the interior of $R_k$, or $C_k$ is a lower DCC, in which case the vertices of $g_2$ above $\ell$ are not included in $R_k$. We have thus shown that DOUBLEUP cannot have used $w$.

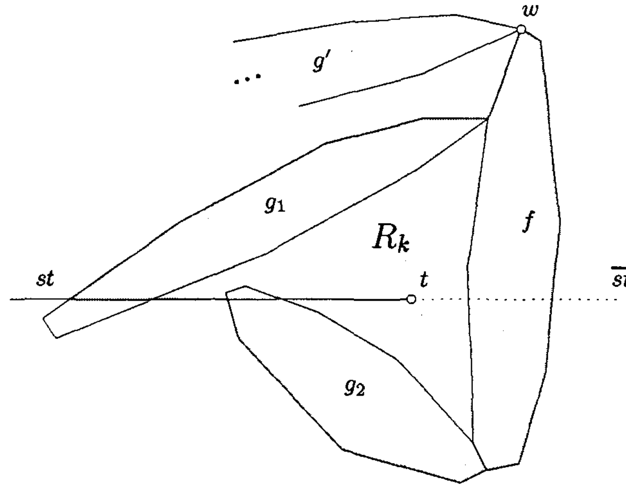


Figure 4.17: DOUBLEUP cannot have used $w$ before arriving at $C_k$, for then $g' \cap \ell \prec g_1 \cap \ell$, contradicting the minimality of $g_1$. Compare to Figure 4.16.

We omit the similar arguments for the other two agents. □

We have finally built enough machinery to finish the proof of the theorem with ease.

**Claim 4.5** *The walks described by the three agents are mutually internally vertex-disjoint.*

*Proof.* Each agent's walk $W$ naturally decomposes into a sequence of subwalks $W_0, W_1, \ldots, W_k$ where $W_j$ is the subwalk of $W$ traversed during the $j$th phase, $0 \leq j \leq k$. By Claims 4.3 and 4.4 we immediately see that if two agents' walks have a vertex $v$ in common, then $v$ must have been used by both agents in the $j$th phase, and before the end of the phase, for some $j$. If $v$ is on the boundary of $R_j$ (in which case $j > 0$) then by Claim 4.4, both agents must use $v$ during the initial adjustments. But induction on Claim 4.2 and the adjustments specified in DCC-ADJUST clearly show that the agents' adjustments constitute disjoint walks. Therefore $v$ is in the interior of $R_j$ and both agents use $v$

after the adjustments. From the discussion of the problem so far, it should be obvious that this is impossible: the double agents cannot use the same vertex, and the solitary agent adjusts to SCCs. □

This completes the proof of Theorem 4.1. ■

Finally, observe that DOUBLE-CROSSING runs in polynomial time: there are clearly finitely many DCCs, and each agent takes polynomial time to move from one DCC to the next (or to $t$).

## 4.4 The DOUBLE-CROSSING algorithm in $\mathbb{R}^2$

As mentioned in the previous section, the DOUBLE-CROSSING algorithm was presented for a graph embedded on a sphere mainly for convenience. The procedures in the algorithm only calculate angles and intersections, which can obviously be accomplished in $\mathbb{R}^2$.

The circle $\ell$ in $\mathbb{R}^3$ corresponds to the line segment $\ell$ (abusing notation here and subsequently) in $\mathbb{R}^2$ through $st$ extending as far as the outer face $f_O$. However, in $\mathbb{R}^2$, $\overline{st} := \{\ell \setminus st\} \cup \{s,t\}$ is a union of two (possibly trivial) line segments.

We can define lexicographic orderings $\prec$ on $st$ and $\overline{st}$ analogous to those in $\mathbb{R}^3$. Take $xLy$ to read "$x$ is to the left of $y$"; then define $\prec$ as follows:

- for $x, y \in st$, $x \prec y$ if $xLy$
- for $x, y \in \overline{st}$, $x \prec y$ if one of the following three conditions holds:

  1. $xLs$, $yLs$, and $yLx$;
  2. $xLs$ and $tLy$; or
  3. $tLx$, $tLy$, and $yLx$.

The proof of correctness for the algorithm in $\mathbb{R}^2$ is essentially the same as the proof in the previous section. However, in that section we took the liberty of viewing DCCs as being centred around $t$; thus, for example, the definition of the active region $R_k$ was simple. In the plane, however, we do not have the same luxury. The outer face is fixed,

and must be treated with special cases: compare Figures 4.18, and 4.19, for example, which both show DCCs.

Despite the apparent added complexity in the plane, the DOUBLE-CROSSING algorithm (with data structures and procedures modified in the obvious ways for $\mathbb{R}^2$) will still work. The facial structure is fundamentally the same as that on the sphere, through the correspondence of $st$, $\overline{st}$, and $\prec$. The only extra challenge in $\mathbb{R}^2$ is surviving the tedium of working through all cases for the positions of DCCs in the proof of correctness.

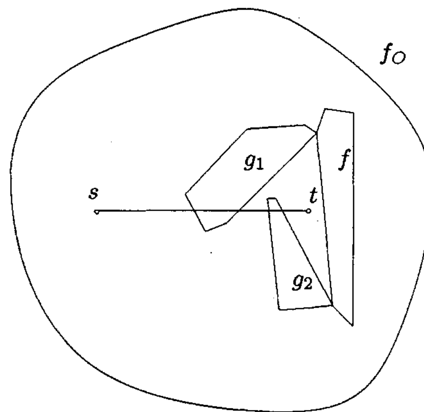


Figure 4.18: A DCC $(f, g_1, g_2)$ in $\mathbb{R}^2$. Compare with Figure 4.5.

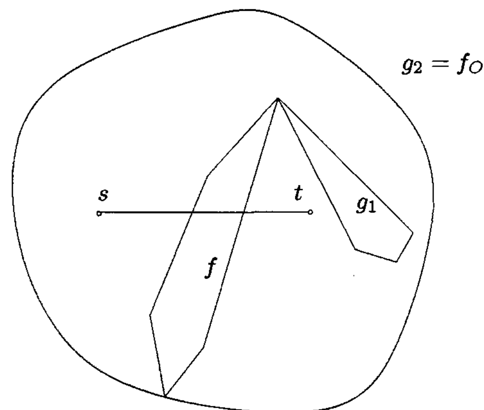


Figure 4.19: Another DCC $(f, g_1, f_O)$ in $\mathbb{R}^2$. The outer face is the lower face of the DCC.

## 4.5 Improving the algorithm

There are three simple modifications we can make to the algorithm to improve its efficiency. The first takes advantage of the information provided by the scouts to accelerate the face routing of all agents; the second eliminates unnecessary scout activity; and third is specifically an improvement for the solitary agent in $\mathbb{R}^2$ during the 0th phase (here we mean the phase only as far as that agent is concerned).

*Improvement 1.* Without loss of generality, assume that DOUBLEUP is travelling along $st$. If the agent is on a vertex $v$ incident to at least two $st$-crossing faces, and is not on a DCC, then the agent can skip forward in its face routing and set *currentFace* to be the farthest $st$-crossing face incident to $v$ with respect to $\prec$. This is shown for DOUBLEUP in Figure 4.20. The same modification accelerates the face routing of all agents; note that it does not affect the order in which the agent detects DCCs, by familiar arguments.

To accelerate the face routing in this manner requires only a trivial modification for the scouts: since the agent's child scout is already searching the faces incident to the current vertex, the scout can simply devote an additional memory slot to keeping track of the farthest $st$-crossing (or $\overline{st}$-crossing, depending on *doubleDirection*) face.

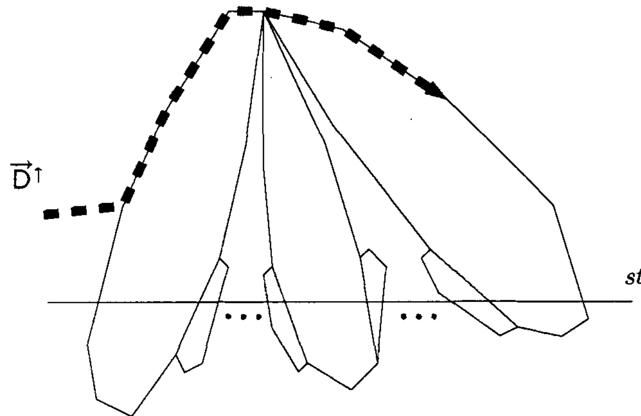


Figure 4.20: In general, every agent can accelerate its face routing using information provided by the scouts.

*Improvement 2.* We can eliminate many unnecessary face traversals for the scouts at every iteration by appealing to the convexity of the faces. Suppose a scout is created at a vertex $v$ to search the face $f$ containing $u$, $v$, and $w$, where $vu$ and $vw$ are radially

adjacent edges. Then, since the scouts are searching for faces intersecting $\ell$ within the active interval, we can immediately eliminate $f$ if $cone(u, v, w) \cap L \cap activeInterval = \emptyset$, where $L$ is either $st$ or $\overline{st}$ according to the ancestor agent's parameters and the type of face being sought. Figure 4.21 shows an example.

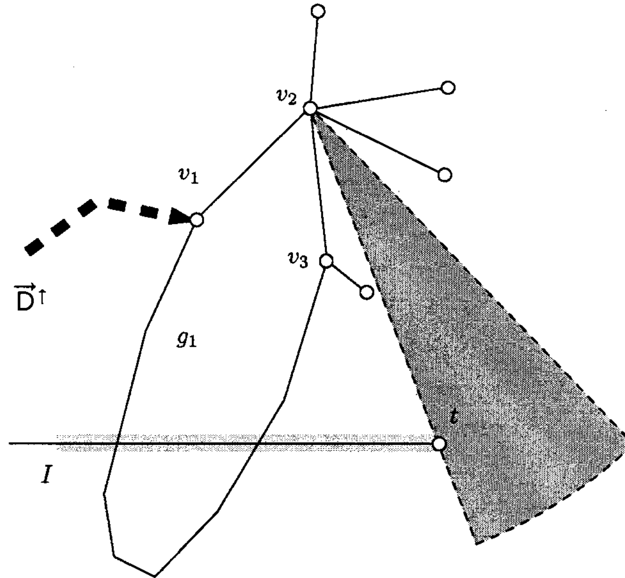


Figure 4.21: DOUBLEUP has arrived at $v_1$, and its child scout $S_1$ has found the $st$-crossing face $g_1$, a potential upper face; $S_1$ is currently at $v_2$. Only one face, $f$, incident to $v_2$ can "see" part of active interval $I$ intersecting $\overline{st}$, so $S_1$ only creates a scout $S_2$ to search the potential central face $f$. If $S_2$ determines that $f$ does not intersect $I \cap \overline{st}$, then $S_1$ continues to $v_3$, which is incident to two potential central faces.

*Improvement 3.*

The third modification is a natural response to the solitary agent's initial behaviour (*i.e.*, until it arrives at a DCC) in the plane. According to the algorithm, we begin by sending the solitary agent away from the destination $t$. Barring encounters with DCCs, it travels away from $t$ (in a strict geometric sense; according to $\prec$ it is approaching $t$) until hitting $f_O$, at which point it traverses half the perimeter of the graph before eventually arriving at $t$. This is terribly inefficient, and the obvious question is: can we find a better path for the solitary agent?

Indeed we can, in most cases. Modify the solitary agent's behaviour so that it initially travels towards $t$ in the same direction as the double agents, using the faces

immediately below the $st$-crossing faces; see Figure 4.22. Denote the family of such faces by $H$. The solitary agent can clearly detect faces in $H$ using scouts, and will run into a problem only if $f_O \in H$ or if a face in $H$ contains a vertex above $\ell$ incident to an $st$-crossing face, since DOUBLEUP uses precisely those vertices. But then we are simply describing SCCs and DCCs – the agents adjust, and continue as usual. Note that this modification applies only to the initial solitary agent, and not to an agent switching into SOLITARY mode during an adjustment at a DCC. Also note that the decision to traverse faces below $st$-crossing faces is arbitrary; the modified agent could instead traverse faces above $st$-crossing faces.

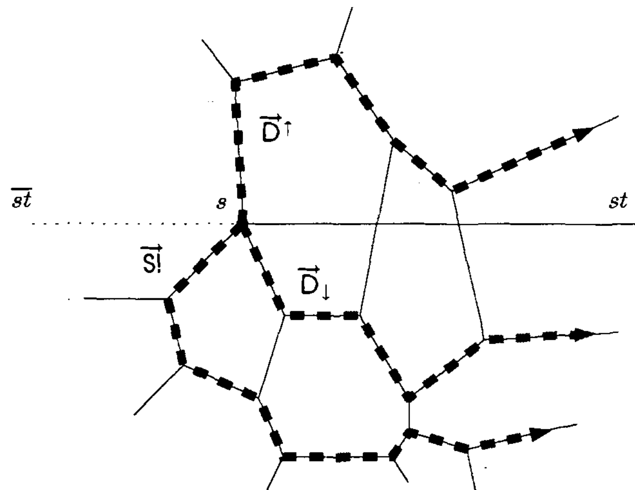


Figure 4.22: The improved initial solitary agent S! uses faces below the $st$-crossing faces and travels in the same direction as the double agents.

# Appendix A

# Memory requirements

The following table lists the memory requirements, in number of vertices, for the algorithms presented in this thesis. The well-known algorithms FACE and GFG are shown first for comparison.

| Algorithm name | Type | Class of graphs | Memory requirement |
|---|---|---|---|
| FACE | routing | planar | 3 |
| GFG | routing | unit disk, planar | 4 |
| QUASI-PLANAR | routing | quasi-planar | 3 |
| QFQ | routing | unit disk, quasi-planar | 4 |
| QUASI-POLYHEDRAL | routing | quasi-polyhedral | 4 |
| SPIRAL | geocasting | quasi-planar | 4 |
| DOUBLE-CROSSING | disjoint routing | convex embedding | 3 per agent |

Also note that the agents in the DOUBLE-CROSSING algorithm also use scouts which each store two vertices in memory.

# Bibliography


[BFN01] L. Barrière, P. Fraigniaud, and L. Narayanan. Robust position-based routing in wireless ad hoc networks with unstable transmission ranges. In *DIALM '01: Proceedings of the 5th international workshop on Discrete algorithms and methods for mobile computing and communications*, pages 19–27, New York, NY, USA, 2001. ACM Press.

[BM99] P. Bose and P. Morin. Online routing in triangulations. In *ISAAC: 10th International Symposium on Algorithms and Computation (formerly SIGAL International Symposium on Algorithms), Organized by Special Interest Group on Algorithms (SIGAL) of the Information Processing Society of Japan (IPSJ) and the Technical Group on Theoretical Foundation of Computing of the Institute of Electronics, Information and Communication Engineers (IEICE))*, 1999.

[BM01] P. Bose and P. Morin. Competitive online routing in geometric graphs. In *Proc. 8 Information and Communication Complexity*, pages 35–44. Carleton University Press, 2001.

[BMB+00] P. Bose, P. Morin, A. Brodnik, S. Carlsson, E. Demaine, R. Fleischer, J. Munro, and A. Lopez-Ortiz. Online routing in convex subdivisions. In *International Symposium on Algorithms and Computation*, pages 47–59, 2000.

[BMSU01] P. Bose, P. Morin, I. Stojmenovic, and J. Urrutia. Routing with guaranteed delivery in ad hoc wireless networks. *Wireless Networks*, 7(6):609–616, 2001.

[dBvKOS00] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwartzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 2000.

[DJ89] G. Das and D. Joseph. Which triangulations approximate the complete graph? In *Proceedings of the international symposium on Optimal algorithms*, pages 168–192, New York, NY, USA, 1989. Springer-Verlag New York, Inc.

[GS69] K. Gabriel and R. Sokal. A new statistical approach to geographic variation analysis. *Systematic Zoology*, 18:259–278, 1969.

[KGKS05] Y. Kim, R. Govindan, B. Karp, and S. Shenker. On the pitfalls of geographic face routing. In *DIALM-POMC '05: Proceedings of the 2005 joint workshop on Foundations of mobile computing*, pages 34–43, New York, NY, USA, 2005. ACM Press.

[KMS06] E. Kranakis, T. Mott, and L. Stacho. Online routing in quasi-planar and quasi-polyhedral graphs. In *Fourth IEEE International Conference on Pervasive Computing and Communications: WORKSHOPS*, pages 426–430, 2006.

[KRD06] S. Kumar, V. Raghavan, and J. Deng. Medium access control protocols for ad-hoc wireless networks: A survey. *Ad Hoc Networks*, 4:326–358, May 2006.

[KSU99] E. Kranakis, H. Singh, and J. Urrutia. Compass routing on geometric networks. In *Proc. 11 th Canadian Conference on Computational Geometry*, pages 51–54, Vancouver, August 1999.

[LL99] C. Lin and J. Liu. QoS routing in ad hoc wireless networks. *IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS*, 17(8), August 1999.

[Mor01] P. Morin. *Online Routing in Geometric Graphs*. PhD thesis, Carleton University, School of Computer Science, 2001.

[PH99] M. Pearlman and Z. Haas. Determining the optimal configuration for the zone routing protocol. *IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS*, 17(8), August 1999.

[PR02] P. Paul and S. Raghavan. Survey of QoS routing. In *ICCC '02: Proceedings of the 15th international conference on Computer communication*, pages 50–75, Washington, DC, USA, 2002. International Council for Computer Communication.

[SR34] E. Steinitz and H. Rademacher. *Vorlesungen über die Theorie der Polyeder unter Einschluß der Elemente der Topologie*. Springer-Verlag, Berlin, 1934.

[Tut63] W. Tutte. How to draw a graph. In *Proc. London Math. Society*, volume 13, pages 743–768, 1963.