Local routing algorithms for the congested half-Θ_6-graph

RACHEL KWOK
SID: 510463067

Supervisor: Dr. André van Renssen

This thesis is submitted in partial fulfillment of
the requirements for the degree of
Bachelor of Science and Bachelor of Advanced Studies (Honours)

School of Computer Science
The University of Sydney
Australia

1 November 2024

THE UNIVERSITY OF
SYDNEY

Student Plagiarism: Compliance Statement

I Rachel Kwok certify that:

I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure;

I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to the University commencing proceedings against me for potential student misconduct under Chapter 8 of the University of Sydney By-Law 1999 (as amended);

This work is substantially my own, and to the extent that any part of this work is not my own I have indicated that it is not my own by acknowledging the source of that part or those parts of the work.

Name: Rachel Kwok

Signature:  **Date**: 01/11/24

Abstract

When congestion is present in a geometric graph, how can we still route efficiently? While this may be a more simplistic problem if we had information about the entire graph, how can effective routing still be achieved when we are constrained to using a constant amount of memory and local information? We approach this problem by studying the half-Θ_6-graph and leverage its geometric properties to route when congestion is present, initially focusing on a congested region confined to a half-plane and later extending to a convex polygon. We contribute deterministic local routing algorithms for both cases. The half-plane routing algorithm is proved to be 4-competitive when routing positively and 4.9-competitive when routing negatively. In addition, we show that no local routing algorithm can do better in the case where our algorithm successfully finds an uncongested $s - t$ path.

We show a negative result when the congested region becomes a convex polygon: no local routing algorithm can do better than an $O(c)$-approximation of the shortest path. Regardless, we designed a deterministic local routing algorithm for finding if an uncongested path exists within some user-defined distance. We show that in cases where it can find an uncongested path, it produces a 13.1-approximation of the shortest uncongested path when the user-defined distance does not exceed $\frac{13.1}{2}$ times the polygon's perimeter and a 17.5-approximation when it does.

This work represents the first time congestion has been studied in the context of Θ-graphs and introduces the first local routing algorithms developed specifically for this problem.

Acknowledgements

To begin, I want to thank Dr. André van Renssen for inspiring my interest in algorithms and being a fantastic supervisor. You have always made me feel like my ideas are valued and I'm very glad to have been able to work with you. I'd also like to acknowledge my family and friends for their endless support, which I could not have gone without. Finally, to the two rainbow lorikeets who have visited me every single day for years: thank you for annoying me and accompanying me on this journey.

CONTENTS

List of Figures

List of Tables

CHAPTER 1

Introduction

Routing in geometric networks is a fundamental problem which has been extensively studied. When information about the entire graph is available, we have a range of graph algorithms at our disposal for finding a path between two vertices. Some standard graph algorithms include Dijkstra's algorithm (Dijkstra, 1959), Breadth-First Search (Moore, 1959) and Depth-First Search (Tarjan, 1971). While simplistic and generally efficient, their Achilles' heel is that they often cannot be applied in an online setting as the network becomes too large to feasibly store. In addition, we typically only have access to local geometric information, such as the immediate neighbourhood of a vertex, to compute the next best move to get to our destination. This requirement for a routing strategy to be local is especially difficult but proved to be possible when we use known geometric properties of the graph.

However, when realistically modelling a network, being able to find *any* path from one vertex to another is the bare minimum. This challenge is compounded when we consider the possibility of network congestion, in which certain network edges and nodes become overloaded with traffic. This may cause delays in message routing and reduces the overall efficiency of the entire network. Hence, designing routing strategies which can still attain acceptable path lengths under congested settings is highly motivated by practical applications such as autonomous vehicle routing and network communications.

1.1 Spanners

A geometric graph G is a graph where its vertices are points in a plane and every edge is weighted by the Euclidean distance between its endpoints. The distance $d_G(u, v)$ is defined as the sum of the weights of the edges in the shortest path between u and v in G. A subgraph H of G is a t-spanner of G, for some constant $t \geq 1$, if for all pairs of vertices u and v, $d_H(u, v) \leq t \cdot d_G(u, v)$. G is referred to as the *underlying graph*. The spanning ratio of H is the smallest t for which it is a t-spanner. Typically,

when referring to spanners, the underlying graph is the complete Euclidean graph. This is because the main motivation behind the use of spanners is being able to approximate the complete Euclidean graph and decrease the number of edges while preserving the length of the shortest path as closely as possible. Some well-known classes of t-spanner networks include Delaunay triangulations and Θ-graphs. Despite having been extensively studied for over 20 years, tight spanning ratios are only known for a select few classes of Delaunay triangulations and Θ-graphs.

1.2 Local Routing

A routing algorithm takes a pair of vertices (s, t) as input and finds a path from s to t in the graph. Formally, a routing algorithm is a deterministic k-local routing algorithm if it only uses information about s, t and the k-neighbourhood of the current vertex to make its forwarding decisions. In addition, it is assumed that the only information stored at each vertex are its immediate neighbours. Such a routing algorithm is said to be c-competitive if the total distance travelled is never more than c times the length of the shortest path between s and t, for all pairs of points. This constant c is otherwise referred to as the *routing ratio*. In this thesis, we use the term *approximation ratio* interchangeably.

1.3 Problem

This thesis focuses on the half-Θ_6-graph, a class of Θ-graph which has a known tight spanning ratio of 2. While a local optimal routing algorithm for the half-Θ_6-graph is known (Bose et al., 2015a), we observe that it does not consider edge weight when deciding which vertex to traverse to next. We simulate congestion on the half-Θ_6-graph by marking some subset of vertices as "congested". Every edge with at least one congested vertex as an endpoint will have its edge weight scaled by a factor of c. This renders the aforementioned algorithm, as well as all known local routing algorithms to be sub-optimal on the half-Θ_6-graph. Naturally, we ask: how can we route locally and *effectively* when congestion is present in the half-Θ_6-graph?

1.4 Contributions

We decompose this problem by considering the congested region when it is first contained within a half-plane and then, contained within a convex polygon. Our main contributions are the design and analysis

of two deterministic local routing algorithms: one for routing around the congested region contained within a half-plane and one for routing around the congested region contained within a convex polygon. Moreover, we show that no local routing algorithm can do better than an $O(c)$-approximation when the congested region is in the shape of a convex polygon, where c is the congestion factor. To the best of our knowledge, this is the first time congestion has been studied in the context of Θ-graphs and these are the first local routing algorithms proposed.

1.5 Outline

In Chapter 2, we delve into existing literature surrounding Θ-graphs, Delaunay triangulations and known local routing algorithms algorithms for them. We also formalise our definition of congestion and examine what work has been done around routing in the presence of network congestion.

We introduce our first local routing algorithm for routing around a congested half-plane in Chapter 3. This is followed by its analysis in Chapter 4, in which we explain some key design decisions and derive its routing ratio.

In Chapter 5, we show a negative result: no local routing algorithm can do better than an $O(c)$-approximation of the shortest path length when the congested region is contained in a convex polygon. Regardless, we introduce our second local routing algorithm, which focuses on determining if there exists a completely uncongested path of a user-designated length. We follow this with some analysis on the quality of the path produced in Chapter 6.

Finally, we conclude our findings in Chapter 7 and suggest ideas for further work.

CHAPTER 2

Literature Review

2.1 Θ-graphs and Delaunay Triangulations

Θ-graphs are a popular geometric graph for modelling wireless network topologies and belong to the family of cone-based spanners. It was introduced independently by Clarkson (1987) and Keil (1988) as a method to approximate the complete Euclidean graph. Θ-graphs are constructed by partitioning the plane around each vertex into k disjoint cones, with the vertex as the apex. The orientation of the cones is the same for every vertex, with each cone being defined by two rays at consecutive multiples of $\theta = 2\pi/k$ radians apart. These cones are oriented such that the bisector of some cone coincides with the vertical half-line that comes from above u and goes through it. Within each cone, we connect u to the vertex v which possesses the smallest projection onto the bisector of the cone (see Figure 2.1). This resulting graph is denoted as the Θ_k-graph.

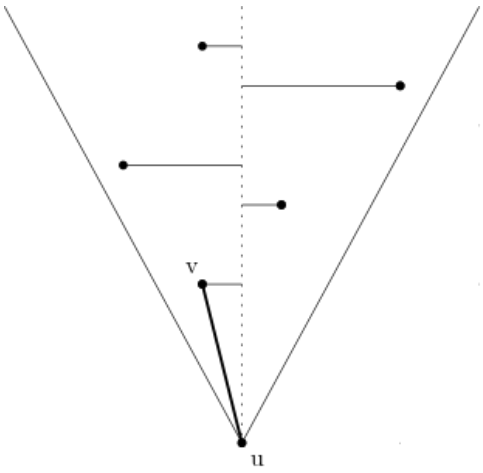


FIGURE 2.1. Vertices projected onto the bisector of a cone with apex u. Vertex v is closest and is connected by an edge to u.

Ruppert and Seidel (1991) found that for $\theta < 2\pi/3$ i.e. there are at least seven cones, the spanning ratio is at most $1/(1 - 2\sin(\theta/2))$. The Θ_4-graph and Θ_5-graph have also been found to be spanners, with a spanning ratio of at most $(1 + \sqrt{2}) \cdot (\sqrt{2} + 36) \cdot \sqrt{4 + 2\sqrt{2}} \approx 237$ (Barba et al., 2013) and $\sqrt{50 + 22\sqrt{56}} \approx 9.960$ (Bose et al., 2015b) respectively. This was recently improved to 17 (Bose et al., 2024) for the Θ_4-graph and 5.70 (Bose et al., 2021) for the Θ_5-graph. On the other hand, Θ-graphs with fewer than 4 cones have been shown to not be spanners.

Bose et al. (2016) improve on the upper and lower bounds on the spanning ratio of 4 broad families of Θ-graphs: Θ_{4k+2}-graph, Θ_{4k+3}-graph, Θ_{4k+4}-graph and Θ_{4k+5}-graph, where $k \geq 1$ is an integer (see Table 2.1). Out of these, it was shown that the Θ_{4k+2}-graph has a tight spanning ratio of $1 + 2\sin(\theta/2)$. This result was obtained by generalising the inductive spanning proof of the half-Θ_6-graph (Bose et al., 2015a), for which the Θ_6-graph and the half-Θ_6-graph share a tight spanning ratio of 2 (Paul Chew, 1989).

	Spanning ratio
Θ_4-graph	17 (Bose et al., 2024)
Θ_5-graph	$\frac{\sin(\frac{3\pi}{10})}{\sin(\frac{2\pi}{5})-\sin(\frac{3\pi}{10})} \approx 5.70$ (Bose et al., 2021)
Θ_6-graph	2 (Paul Chew, 1989)
Θ_{4k+2}-graph	$1 + 2\sin(\theta/2)$ (Bose et al., 2016)
Θ_{4k+3}-graph	$\frac{\cos(\theta/4)}{cos(\theta/2)-\sin(3\theta/4)}$ (Bose et al., 2016)
Θ_{4k+4}-graph	$1 + \frac{2\sin(\theta/2)}{\cos(\theta/2)-\sin(\theta/2)}$ (Bose et al., 2016)
Θ_{4k+5}-graph	$\frac{\cos(\theta/4)}{cos(\theta/2)-\sin(3\theta/4)}$ (Bose et al., 2016)

TABLE 2.1. Current known upper bounds on the spanning ratios for Θ-graphs. Tight spanning ratios are in blue.

Delaunay triangulations are defined by creating an edge between u and v if there exists a circle with u and v on the boundary and no other vertex within. While the exact spanning ratio is unknown, the current best upper bound is 1.998, proven by Xia (2013). There exist many variants of Delaunay triangulations as they can be defined by different distance metrics. Typically, the distance between two points u and v in the plane is defined as $((x_u - x_v)^2 + (y_u - y_v)^2)^{\frac{1}{2}}$. This can be generalised to a family of metrics L_p, where the distance is defined as $((x_u - x_v)^p + (y_u - y_v)^p)^{\frac{1}{p}}$. Naturally, this affects the shape of the empty "circle" i.e under L_1, the shape would become a diamond. Despite knowing Delaunay triangulations using arbitrary convex shapes are spanners, the matter of finding tight spanning ratios is much more challenging. Tight bounds on the spanning ratio are only known when the shape is an equilateral triangle, square, regular hexagon or rectangle (see Table 2.2).

Shape	Spanning ratio
Equilateral triangle	2 (Paul Chew, 1989)
Square/Diamond	2.61 (Bonichon et al., 2015)
Regular hexagon	2 (Perković et al., 2022)
Rectangle	$\sqrt{2}\sqrt{1+A^2+A\sqrt{A^2+1}}$ (van Renssen et al., 2023)

TABLE 2.2. Known tight spanning ratios for Delaunay triangulations, where $A = l/s$ (aspect ratio)

2.1.1 The Half-Θ_6-graph

The half-Θ_6-graph was first introduced by Bonichon et al. (2010), who also contribute that it is exactly the TD-Delaunay graph. The TD-Delaunay triangulation is a variation of the classical Delaunay triangulations, described in Section 2.1, where the empty region is an equilateral triangle. This allows the half-Θ_6-graph to inherit properties of the TD-Delaunay graph, notably its tight spanning ratio of 2 as proved by Paul Chew (1989). This also applies for Θ_6-graphs as the TD-Delaunay graph is a spanning subgraph of it, making the Θ_6-graph and the half-Θ_6-graph one of the few Θ-graphs where tight bounds are known (see Table 2.1).

To construct the half-Θ_6-graph, we first label the cones $\bar{C}_1, C_0, \bar{C}_2, C_1, \bar{C}_0$ and C_2, in anti-clockwise order around u, starting from the positive x-axis (see Figure 2.2). These cones are then divided into *positive* and *negative* cones, with C_0, C_1 and C_2 as *positive* and others as *negative*. While a traditional Θ-graph connects u to the vertex which has the closest projection onto the bisector within *every* cone, the half-Θ_6-graph only connects vertices in the positive cones.

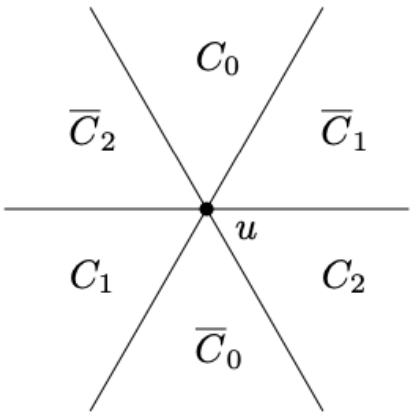


FIGURE 2.2. Cones in the half-Θ_6-graph.

2.2 Routing

2.2.1 Routing in Delaunay Triangulations

There has been much work done on routing on the classical L_2-Delaunay triangulation. Bose and Morin (2004) presented a total of five routing algorithms and proved that they are all undefeated by Delaunay triangulations. These include greedy routing and compass routing, two simplistic and well-known routing strategies. Greedy routing, as its name suggests, always moves to a neighbouring vertex v which minimises the Euclidean distance between v and the destination. Compass routing moves to the neighbouring vertex which minimises the angle between it, the current vertex and the destination. However, both these strategies are not c-competitive as a configuration of points can be contrived to make the path produced arbitrarily large. Evidently, both algorithms do not fully take advantage of the known properties of Delaunay triangulations. Using the upper bound of the spanning ratio found by Dobkin et al. (1990), Bose and Morin (2004) were able to present a $(9(1+\sqrt{5})\pi/2) \approx 45.75$-competitive routing algorithm.

Following Xia's proof of a better upper bound of 1.998 (Xia, 2013) on the classical Delaunay triangulation, Bose et al. (2017) contributed two new routing algorithms with a competitive ratio of 17.982 and were able to reduce the competitive ratio even further to 15.479. Bonichon et al. (2017) were able to improve upon this by generalising Chew's algorithm for the L_1-Delaunay triangulation (Paul Chew, 1989) and presented a routing algorithm with a routing ratio of 5.90. This was further improved by Bonichon et al. (2023) who proposed a routing algorithm with a routing ratio of 3.56

2.2.2 Routing in Θ-graphs

There exists a routing algorithm for Θ-graphs called cone-routing, which always moves to a vertex that is within the same cone as t. This strategy is akin to greedy routing but has seen markedly better success, as it was proven to be competitive on Θ_k-graphs for $k \geq 7$. It produces a routing ratio of $1/(1-2\sin(\theta/2))$ (Ruppert and Seidel, 1991). However, for $k \leq 6$, cone-routing produces unbounded routing ratios and thus, fails to be competitive (Bose et al., 2020). Bose et al. (2024) introduced a local routing algorithm for directed and undirected Θ_4-graphs with a routing ratio of at most 17, being the first competitive routing algorithm for $k = 4$. However, while competitive, little is known about its optimality.

2.3 Congestion

Routing in the context of congestion has remained an expansive domain of interest in network communications. There currently exist many different ways to formulate this problem and define what congestion is. For example, Banner and Orda (2005) formally define the *network congestion factor* as the maximum $\frac{f_e}{c_e}$ over all edges $e \in E$, where f_e is the flow going through that edge and c_e is the edge capacity. This metric is known to provide a good indication of the overall congestion present in the network. Hence, they formulated this problem as an optimization problem of minimizing network congestion. A common approach to handling this is dividing traffic over multiple paths, otherwise known as multipath routing. While it has demonstrated effectiveness in reducing congestion, the algorithm presented by Banner and Orda (2005) for this multipath route uses centralized routing of the entire network topology. Xin et al. (2009) also present a multipath routing strategy, with routing decisions made locally rather than using centralized information. This algorithm is reliant on neighbouring nodes communicating the congestion status to one another to redistribute flows. Evidently, the congestion within this context is more so a measure of the state of the entire network and thus, the most common proposed approaches do not focus on optimising a single path.

When reframing the problem such that congestion is already present within the network, which is one of the assumptions we make for the problem studied in this thesis, popular congestion-aware routing algorithms still place a focus on network design rather than designing the forwarding decisions from the perspective of traversing through the network. For example, the DyAD algorithm (Hu and Marculescu, 2004) has each vertex serve as a router that switches between two states depending on the congestion status of its neighbours. When it encounters congestion, it would enter its adaptive routing state. Within this state, each vertex uses an algorithm called odd-even routing (Chiu, 2000) to move the package along. This moves away from making assumptions about the nature of the graph and using any geometric properties to route. In addition, local congestion-aware routing algorithms are typically designed for two-dimensional meshes, largely due to their structural regularity (Jiang, 2005). As a result, these routing algorithms fail to generalise to networks with different structures.

We observe that the most similar model to how we modelled congestion is present in urban planning literature. A classical problem is finding the shortest path with the presence of some special regions, called barriers. In some instances, travel may be permitted through these barriers but at a higher cost. These barriers are typically used to model areas where travelling or construction is not prohibited but best avoided, such as natural landforms or residential areas. However, algorithms for this problem such

as those proposed by Lozano-Pérez and Wesley (1979) and Xiong and Schneider (1992) do not operate on any fixed graph and hence, allows their algorithm the flexibility to traverse to any point on the plane to avoid these barriers.

2.4 Preliminaries

2.4.1 Congestion

We now formalise our definition of congestion. To begin approaching our problem, we make the assumption that there exists a congested region in the half-Θ_6-graph where all vertices which lie within that region are considered "congested". For every edge (u, v), if either u or v is a congested vertex, its edge weight is updated to $c|uv|$. Regardless of if one or both endpoints are congested, the edge weight will be scaled by this factor of c. We refer to c as the congestion factor and it is constant across all congested vertices.

We assume that both s and t are not within the congested region. This is because there exist instances where if we are forced to traverse into the congested region, any attempt to avoid the congested region will be futile.

2.4.2 Routing in the Half-Θ_6-graph

The half-Θ_6-graph is one of the few graphs where the tight spanning ratio and an associated optimal routing algorithm is known. Surprisingly, there exists a separation between its spanning ratio and the best achievable routing ratio. While it has a spanning ratio of 2, the optimal routing algorithm is guaranteed to find a path which is at most $\frac{5}{\sqrt{3}}$ times the Euclidean distance between the source and the destination (Bose et al., 2015a). Throughout this thesis, we will refer to this algorithm as Bose et al.'s algorithm.

We hereby summarise how the algorithm operates. The algorithm first makes two distinctions: *routing negatively* and *routing positively*. We say that we are *routing positively* when t is in a positive cone of s and *routing negatively* when t is in a negative cone of s. Given two vertices u and v where v is in a positive cone of u, T_{uv} is the canonical triangle defined by the cone of u that contains v and a line through v that is perpendicular to the bisector of the cone (see Figure 2.3). Vertex u is situated at the apex of T_{uv}.

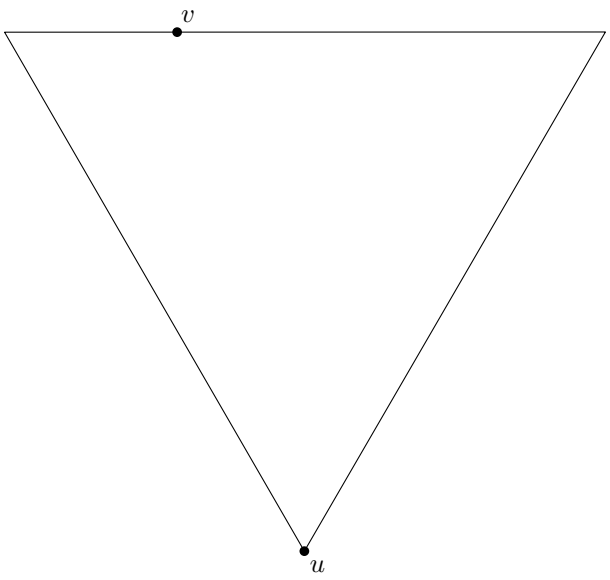


FIGURE 2.3. The canonical triangle T_{uv}.

The algorithm defines one case for routing positively and three cases for routing negatively. At every step, the algorithm will only take vertices which are in the canonical triangle of the current vertex and t.

2.4.2.1 Positive routing

We assume without loss of generality that t is in cone C_0^s. By the construction of the half-Θ_6-graph, the canonical triangle of s and t, T_{st}, will only intersect C_0^s out of all the cones of s. As it only intersects with a single cone, there is only one edge to follow within the canonical triangle and the routing algorithm will follow it. It is shown that there exists an instance where the length of the path followed by any routing algorithm is at least $(\sqrt{3}\cos(\alpha) + \sin(\alpha)) \cdot |st|$ when routing from u to v, α being the angle t creates against the bisector of C_0^s (see Figure 2.4). We can move u as close to the left corner of C_0^s as possible. When $\alpha = \pi/6$, we maximise $|ut|$ and the overall distance travelled, producing a routing ratio of 2. A matching upper bound is also given, showing that this is a tight routing ratio.

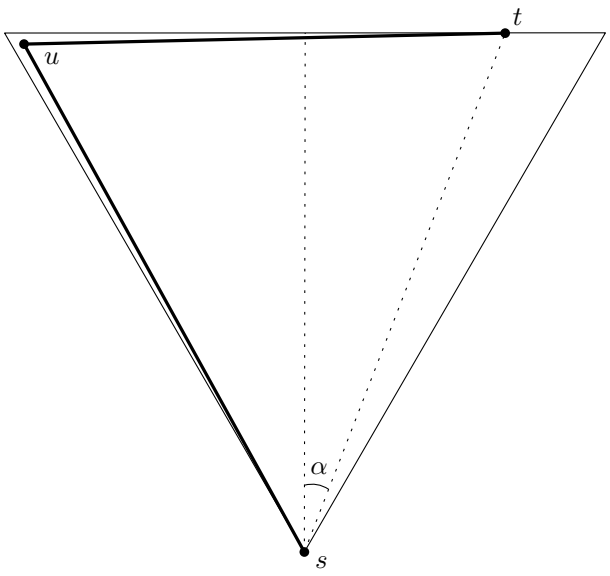


FIGURE 2.4. The lower bound instance for positive routing.

2.4.2.2 Negative routing

We assume without loss of generality that t is in cone $\bar{C}_0^s$ and s is in C_0^t. Hence, T_{ts} intersects $\bar{C}_0^s$, C_1^s and C_2^s. This creates 3 different regions, X_0, X_1 and X_2 (see Figure 2.5). The routing algorithm then selects an edge in T_{ts} based on the emptiness of X_0, X_1 and X_2:

- If both X_1 and X_2 are empty, there must be an edge in X_0. If there are multiple edges in X_0, the algorithm favours staying close to the largest empty side of T_{ts}.
- If exactly one of X_1 and X_2 is empty, choose the edge in X_0 which is close to the empty side of T_{ts}. If there are no edges in X_0, the algorithm would follow the single edge in either X_1 or X_2, whichever is non-empty.
- If X_1 and X_2 are both not empty, follow an arbitrary edge in X_0. If no such edge exists, follow the edge that is in the smaller of X_1 or X_2.

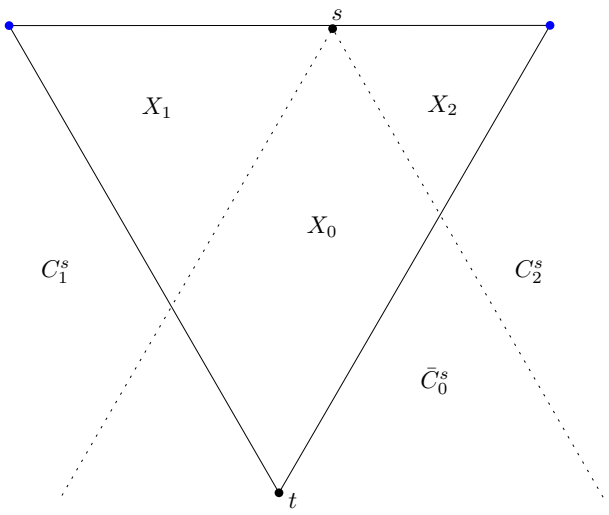


FIGURE 2.5. Regions when routing negatively.

To summarise, our algorithm will choose a vertex in $\bar{C}_0^s$ whenever possible and favour staying close to the largest empty side of T_{ts}.

Analogous to the analysis for positive routing, there exists an instance where the length of the path followed by any routing algorithm is at least $(5/\sqrt{3}\cos(\alpha) - \sin(\alpha)) \cdot |st|$ (see Figure 2.6). In the worst-case, s would be as close to the bisector of C_0^t as possible as that would maximise the distance it has to travel to the rightmost corner. In the configuration depicted in Figure 2.6, it is locally impossible to determine which step is correct from s. Thus, by substituting $\alpha = 0$, we obtain a routing ratio of $\frac{5}{\sqrt{3}}$. A matching upper bound is also provided. This is an especially interesting result as the half-Θ_6-graph has a known tight spanning ratio of 2, making it the first graph where there exists a separation between the spanning and routing ratio.

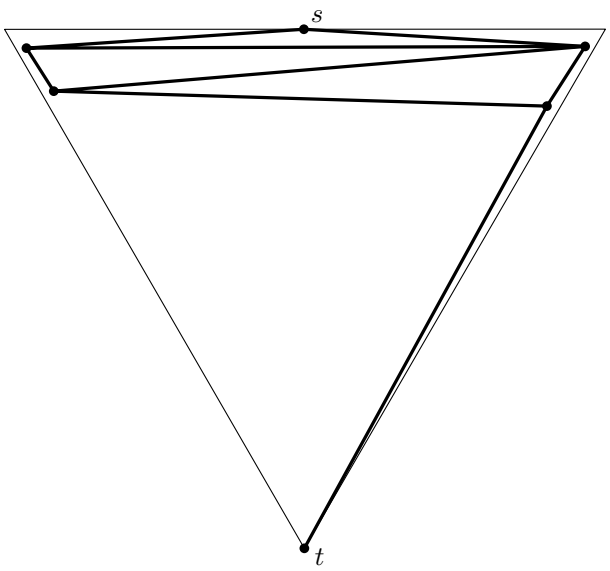


FIGURE 2.6. The lower bound instance for negative routing.

2.4.3 Useful Lemmas

We will now introduce a few useful lemmas which come from the work of Bose et al. (2015a). Moreover, at the end of this section, we briefly outline a popular search strategy which we will use as part of our own algorithm.

One of the main results from Bose et al.'s algorithm is the following:

LEMMA 1. *Given the canonical triangle of s and t, the length of any $s-t$ path contained within it is bounded by $2|st|$ when routing positively and $\frac{5}{\sqrt{3}}|st|$ when routing negatively.*

This lemma can also be rewritten in terms of the side of the canonical triangle of s and t instead of the Euclidean distance:

LEMMA 2. *Given the canonical triangle of s and t, the length of any $s-t$ path contained within it is bounded by $2X$ when routing positively and $2.5X$ when routing negatively, where X is the side length of the canonical triangle of s and t.*

The charging argument for Bose et al.'s routing algorithm is made by using a potential function ϕ. Each step taken in the canonical triangle of s and t will be paid for with the potential, effectively bounding the path length. Let us consider the canonical triangle of s and t, which has vertices labelled as s, a and b.

When routing positively, $\phi = |sa| + \max(|at|, |tb|) \leq 2X$ (see Figure 2.7).

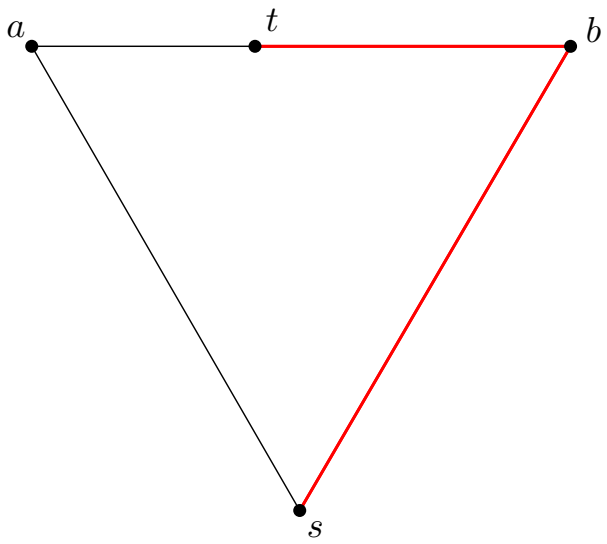
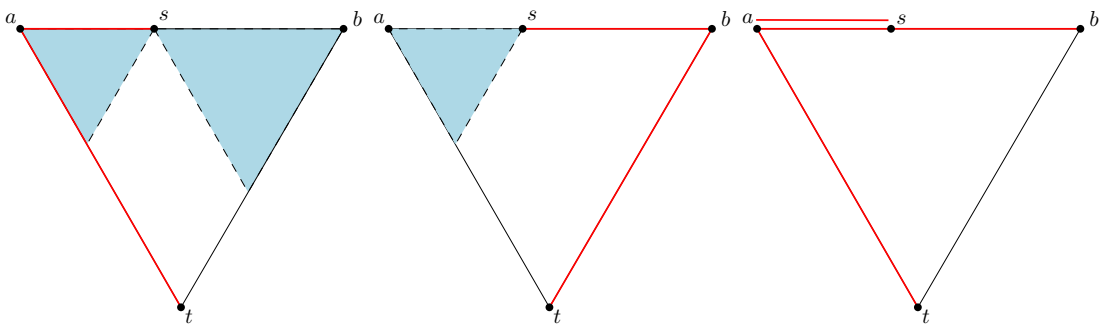


FIGURE 2.7. The potential ϕ for positive routing, shown in red.

When routing negatively, we note that there are three cases (see Section 2.4.2.2). Bose et al. (2015a) define three distinct potential functions:

- If both region X_1 and X_2 are empty, $\phi = |ta| + \min(|as|, |sb|)$.
- If either X_1 or X_2 is empty, let x be the corner contained in the non-empty area and $\phi = |ta| + |sx|$.
- If neither X_1 nor X_2 is empty, $\phi = |ta| + |ab| + \min(|as|, |sb|)$.

In all three cases, we observe that ϕ can be bounded by $2.5X$. This is visualised in Figure 2.8.



(a) Both X_1 and X_2 are empty. (b) Either X_1 or X_2 is empty. (c) Both X_1 and X_2 are not empty.

FIGURE 2.8. The potential ϕ for negative routing. The blue shaded region represents emptiness.

Finally, we observe the following:

LEMMA 3. *If a vertex v can be reached by taking only vertices in a positive cone at each step from u, the path from u to v is bounded by the side length of T_{uv}.*

This insight stems from looking at the charging argument made to bound the path length when routing positively. When Bose et al.'s algorithm takes a step in the positive cone, we can bound every step with the side length of the canonical triangle. This is shown in Figure 2.9, in which v and w are both in a positive cone of u. We observe that taking a step in the positive cone will be charged to the side of T_{uv}. Once we are at w, any step in T_{wv} can also be charged to the same side of T_{uv}.

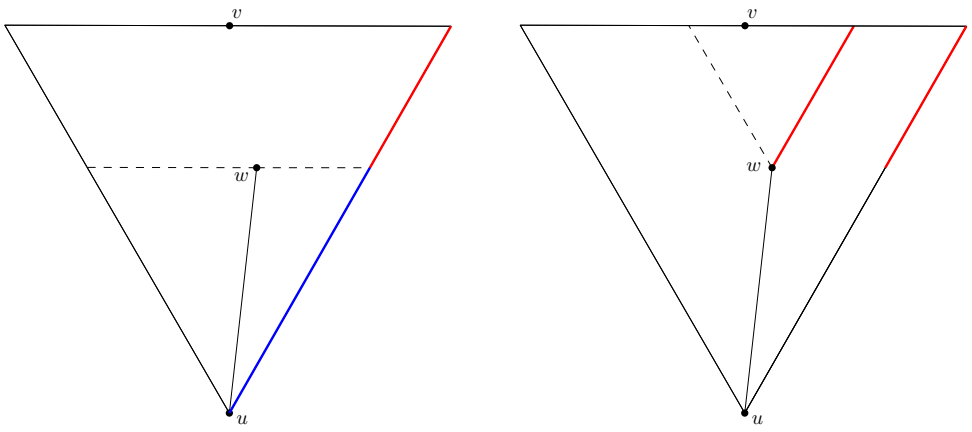


FIGURE 2.9. Steps in a positive cone will always be charged to the same side of the canonical triangle. The red represents the remaining potential and blue represents the potential which was charged from taking the step uw.

In addition, we introduce a popular search strategy coined as the linear spiral search (Baeza-Yates et al., 1993), which be implemented as part of our algorithm in Chapter 5. This search strategy is designed for finding a point on a line which is n steps away. However, we are unaware of which direction along the line the target point is. The linear spiral search algorithm is defined as starting from the origin, walking 1 unit of distance to the right, then returning to the origin and walking 2 units to the left and so on. Each time it returns to the origin, the distance it searches on the following side is doubled. Hence, Baeza-Yates et al. (1993) bounded the total distance travelled with $2(\sum_{i=0}^{\lfloor \log(n) \rfloor + 1} 2^i) + n = 9n$ and showed that this algorithm is optimal.

CHAPTER 3

Routing around congested half-planes

In this chapter, we propose a deterministic local routing algorithm for routing around congested regions contained within a half-plane. We define "seeing" a vertex v as having an edge to v from our current vertex. To briefly summarise, the algorithm is split into three phases: routing, search and return. The routing phase handles uncongested sections of the graph and instances where the algorithm has decided that traversing through the congested region is the best option. The search phase operates by allocating an exploration budget if we see a congested vertex that Bose et al.'s algorithm would go to. This exploration budget is a set distance we can explore before we conclude that it is not reasonable to traverse any further. This budget is updated, depending on the vertices we see along our exploration path. Finally, the return phase is used when we exhaust our exploration budget or if there are no viable vertices to further explore. This returns us to the best vertex to enter the routing phase and begin routing to t again.

3.1 Routing Phase

In uncongested sections of the graph, our algorithm will follow the routing algorithm proposed by Bose et al. (2015a). It will enter the search phase (described in Section 3.2) if it encounters a vertex in the congested region which Bose et al.'s algorithm would choose to traverse to.

3.1.1 Congested routing

This is a variant of the regular routing phase, which we would enter if our algorithm was unable to find another path and commits to traversing through the congested region. Similarly, it follows Bose et al.'s algorithm and checks if we have traversed to a vertex outside of the congested region at each step. If we have, we move back to the regular routing phase so that we can potentially re-enter the search phase.

3.2 Search Phase

Let s be the current vertex we are at and u be the congested vertex that Bose et al.'s algorithm would select. Without loss of generality, let the half-plane defining the congested region intersect C_0^s to the left of s. Upon seeing u, a limit is set on the distance we can travel to search for an alternative path which avoids as much of the congested region as possible. Our algorithm will keep track of a few variables, alongside the positions of s and t:

- $\mathcal{R}$: the remaining budget left.
- $\mathcal{I}$: the total exploration budget allocated. This will be used to calculate how much additional budget should be added to $\mathcal{R}$ once we gain more information about the path in the canonical triangle of s and t.
- λ_{start}: the first uncongested vertex seen in the canonical triangle of s and t (initially set to $\emptyset$).
- λ_{end}: the most recent uncongested vertex seen in the canonical triangle of s and t (initially set to $\emptyset$).
- λ_{dist}: the length of the section of the path which may be uncongested (initially set to 0).
- ϕ: the best vertex to enter the congested region from if we cannot find an alternative path (initially set to s).
- dist(ϕ): our current distance from ϕ.
- u: the most recent vertex seen in the canonical triangle of s and t.

We will refer to the value of $\mathcal{I}$ before making the most recent update to it as $\mathcal{I}'$.

Let v be the vertex we are currently at on our exploration path. We now define the criteria for v to be updated to ϕ. This operation will be referred to as the ϕ-check in future sections. Let vertex w be the highest congested vertex above ϕ when routing positively and the lowest congested vertex below ϕ when routing negatively. If $c|vw| \leq c|\phi w| + \text{dist}(\phi)$, update $\phi = v$ and set $\text{dist}(\phi) = 0$. This condition indicates that entering the congested region at v yields a shorter path than traversing back to ϕ and entering the congested region there. If w does not exist or the condition is not met, we leave ϕ as it is. Consider Figure 3.1 as an example. Assuming that $\phi = s$, we can see that entering the congested region at v is more advantageous than returning back to s and routing to t if we cannot find an uncongested path.

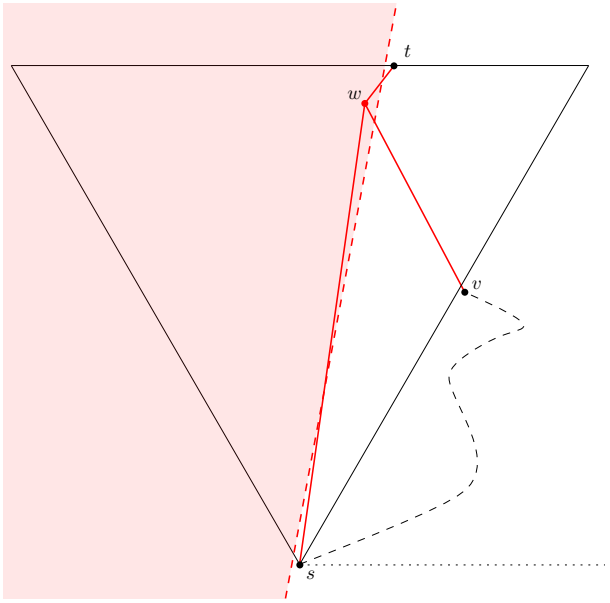


FIGURE 3.1. Example of when entering the congested region at v, a vertex on our algorithm's exploration path, is better than returning to s. The black dashed line represents our algorithm's exploration path and the red edges represent edges connected to a congested vertex.

3.2.1 Positive routing

Without loss of generality, let $t \in C_0^s$. When routing positively, we maintain an additional variable, BACKWARDS, initialised as False, which is used to denote if we are routing away from t. This is necessary as the closest vertex to t in C_2^t may be below s so we may begin by searching upwards to t in $\bar{C}_0^t$ and eventually be forced to backtrack. We define the exploration budget function for positive routing $\mathcal{B}^+(|su|-\lambda_{\text{dist}}) = \frac{2c\sin(\alpha-\frac{\pi}{6})-c^2-1}{2(\sin(\alpha-\frac{\pi}{6})-c)}(|su|-\lambda_{\text{dist}})$, where α is the angle st makes against the right boundary of C_0^s (see Figure 3.2). $\mathcal{I}$ and $\mathcal{R}$ are both initialised as $\mathcal{B}^+(|su| - \lambda_{\text{dist}})$. Broadly speaking, the input parameter in $\mathcal{B}^+$ represents a lower-bound on the length of the congested path in T_{st}. Further details on why $\mathcal{B}^+$ is defined this way is covered in Section 4.2.

3.2.1.1 Exploration path selection

If we are in the search phase when routing positively, that means the only vertex connected to s in T_{st} is in the congested region. This implies we have to leave T_{st} to search for another path. To leave T_{st}, we select the first uncongested vertex in a clockwise direction from the right boundary of C_0^s. Let v be the vertex we are currently at which is outside of T_{st}. If BACKWARDS is False, we select the first uncongested vertex in a clockwise direction from the left boundary of C_0^v which is outside of T_{st}. This

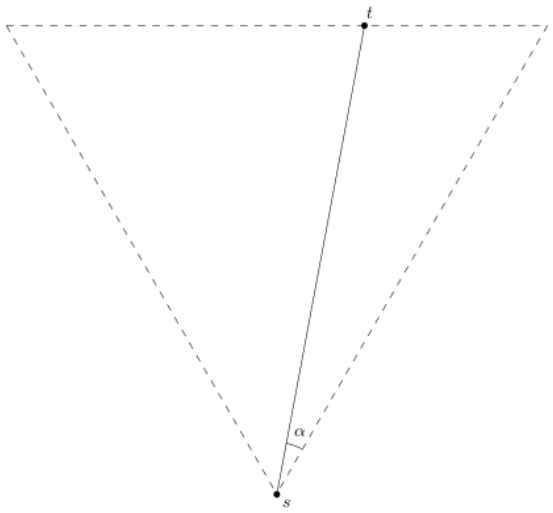


FIGURE 3.2. Visualisation of α, the angle st makes against the right boundary of C_0^s.

is with the exception of that uncongested vertex being in $C_2^t \cap T_{st}$. In this case, we would choose to traverse to it.

If we see a vertex in C_2^t and there are still uncongested vertices above us in either C_0^v or $\bar{C}_2^v$, we should continue traversing upwards first. This may include entering T_{st}. As an example, let us assume we are at vertex v in Figure 3.3. Vertex $w \in C_2^t$ could be placed arbitrarily far away, causing us to run out of exploration budget when we traverse to it. As a result, it would be unwise to immediately traverse to any vertex in C_2^t if there exist unexplored uncongested vertices above us.

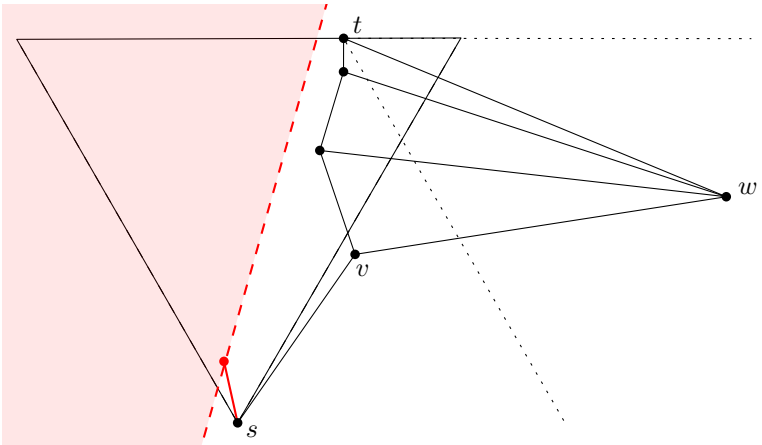


FIGURE 3.3. Example of when it would be unwise to immediately traverse to vertex $w \in C_2^t$. There exists a congested vertex in C_0^s.

If there does not exist an uncongested path when we are in T_{st}, we set BACKWARDS to be True and begin searching backwards by taking vertices in C_2^v only.

If BACKWARDS is already set to True, we only select vertices in C_2^v. If no such vertex exists or the edge to this uncongested vertex has length greater than $\mathcal{R}$, we enter the return phase (described in Section 3.3). There are a few options for where the first vertex in a clockwise direction is located with respect to v:

(1) If it is in C_0^v or $\bar{C}_1^v$, proceed as we are making positive progress towards t.

(2) If it is in in C_2^v, this indicates that there are no uncongested vertices to further explore in either C_0^v or $\bar{C}_1^v$. This now breaks into two subcases:

(2.1) If we are connected to an uncongested vertex in T_{st} and we cannot see a congested vertex in T_{st} which is higher than it, we should first traverse into T_{st} to check for a path. We can make progress towards t by selecting vertices in the direction of t which are closest to the right boundary of T_{st}. An example of this is visualised in Figure 3.4. If no path exists without traversing through the congested region, we set the highest uncongested vertex we can reach as ϕ and enter **Case 2.2**.

(2.2) If we are not connected to any uncongested vertices in T_{st} to further explore, we set BACKWARDS to be True and repeatedly take vertices in C_2^v until we either run out of budget, vertices or make it into C_2^t.

(3) If it is in in $\bar{C}_0^v$ or C_1^v, enter the return phase (described in Section 3.3).

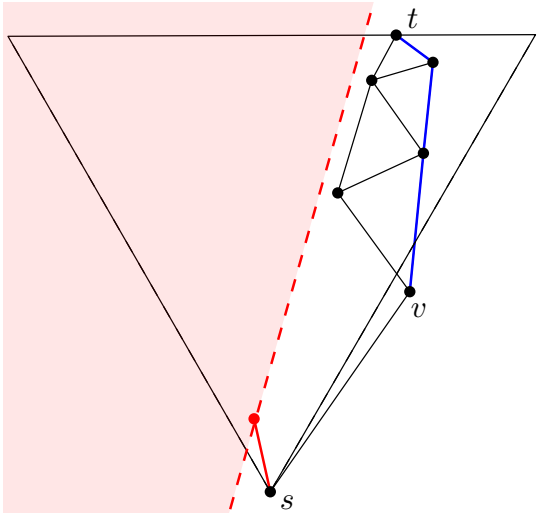


FIGURE 3.4. Example of following vertices closest to the right boundary of T_{st}, where the blueedges represent our algorithm's path.

For every vertex we traverse to, the edge length is then subtracted from $\mathcal{R}$ and added to dist(ϕ). This is with the exception when we are in **Case 2** of above. In addition, at every vertex $v \in \bar{C}_0^t$, we perform the ϕ-check (described in Section 3.2). This is to ensure that we do not miss the possibility that a vertex in $\bar{C}_0^t$ provides a better position to enter the congested region than the previously recorded ϕ (see Figure 3.1).

The above process is repeated until we run out of uncongested vertices to explore. If we are in C_2^t, there are two cases:

(1) If $v \in C_2^t$ and $v \in T_{st}$, we directly enter the routing phase (described in Section 3.1). This is because v must be the highest uncongested vertex in T_{st} and C_2^t is free from congestion when t lies to the right of the congested half-plane.
(2) If $v \notin T_{st}$, we select the first vertex in an anticlockwise direction from the right boundary of $\bar{C}_t^0$ in either $C_1^v, \bar{C}_2^v$ or C_0^v. For every vertex we traverse to, we subtract the edge length to it from $\mathcal{R}$. If no vertex exists within $\mathcal{R}$, we enter the return phase (described in Section 3.3). Note that we may potentially move from this case into the case above if we reach a vertex in T_{st}.

If we are connected to t at any point in our exploration, we traverse directly to t.

3.2.1.2 Budget update

Let v be the vertex we are currently at. Alongside traversing to each vertex, we make updates to $\mathcal{I}$ and $\mathcal{R}$ depending on the vertices we see in $\bar{C}_2^v$ or C_0^v as we explore. This is to get a lower-bound on the path in T_{st}. The general intuition is to dynamically reallocate our remaining budget $\mathcal{R}$ as we gain more information on the path. Recall that we save a variable u which is the most recent vertex seen in T_{st}. If a vertex w is below u, that means we have already found a path around w so we can ignore it and continue exploring, following the same criteria outlined in Section 3.2.1.1. After handling the updates to $\mathcal{I}$ and $\mathcal{R}$, we update u to the highest vertex processed so we do not have to reconsider any vertices if we are connected to the same subset of vertices further along our exploration path.

A vertex $w \in \bar{C}_0^t$ is considered unseen if it is above u. There are five cases for unseen vertices in either $\bar{C}_2^v$ or C_0^v:

(1) If v is connected to a congested vertex w, we introduce two subcases:

(1.1) If $\lambda_{\text{start}} = \emptyset$ (see Figure 3.5), this indicates that the path we see in T_{st} is not an uncongested section. We update $\mathcal{I} = \mathcal{B}^{+}(|sw| - \lambda_{\text{dist}})$ and expand our exploration budget by setting $\mathcal{R} = \mathcal{R} + (\mathcal{I} - \mathcal{I}')$.

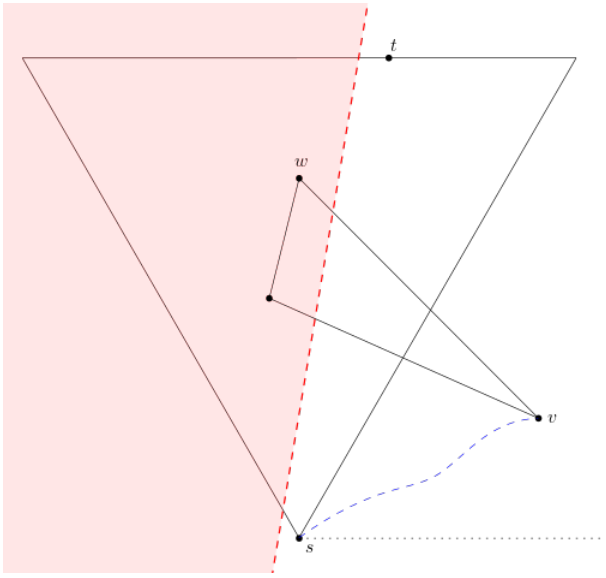
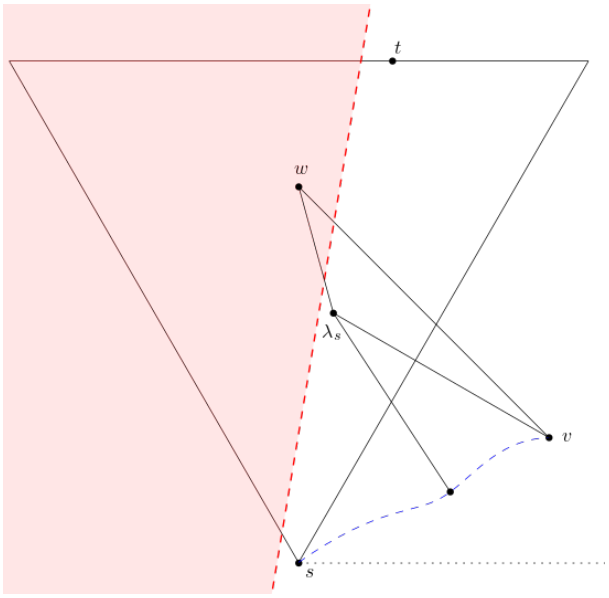


FIGURE 3.5. **Case 1.1**. The blue dashed line represents our exploration path.

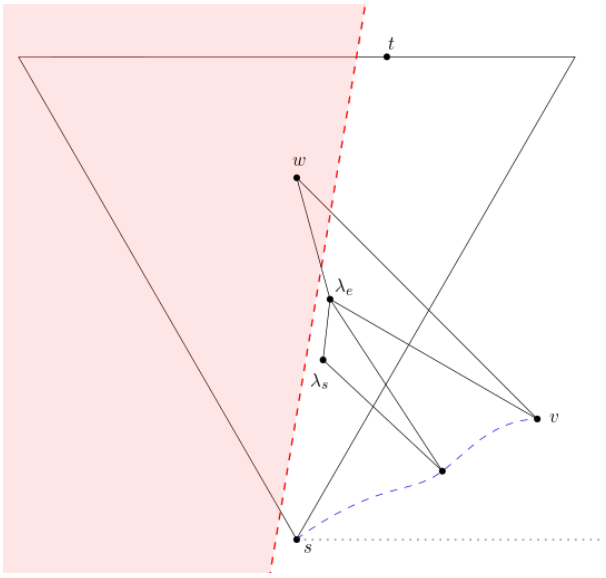
(1.2) If $\lambda_{\text{start}} \neq \emptyset$, check if $\lambda_{\text{end}} = \emptyset$.

If $\lambda_{\text{end}} = \emptyset$, update $\mathcal{I}$ and $\mathcal{R}$ as described in **Case 1.1** and set $\lambda_{\text{start}} = \emptyset$. This indicates that we previously saw a single uncongested vertex in T_{st}. Since it is immediately connected to w which is congested, it is not considered an uncongested section of the path.

If $\lambda_{\text{end}} \neq \emptyset$, set $\lambda_{\text{dist}} = \lambda_{\text{dist}} + |\lambda_{\text{start}}\lambda_{\text{end}}|$. Update $\mathcal{I}$ and $\mathcal{R}$ as described in **Case 1.1** and set λ_{start} and λ_{end} to $\emptyset$. This is done so because seeing a congested vertex implies that the uncongested section of the path, defined by λ_{start} to λ_{end}, has ended. Both of these subcases are visualised in Figure 3.6, in which λ_{start} and λ_{end} are abbreviated to λ_s and λ_e respectively.



(a) **Case 1.2**, where $\lambda_{\text{end}} = \emptyset$. λ_s being connected to another point on the blue dashed line shows that λ_s has already been seen at an earlier vertex along the exploration path.



(b) **Case 1.2**, where $\lambda_{\text{end}} \neq \emptyset$.

FIGURE 3.6. **Case 1.2**

(2) If an uncongested vertex w is in T_{st}, we first check if $|vw| \leq \mathcal{R}+\text{dist}(\phi)$ and update $\phi = w$ and $\text{dist}(\phi) = |vw|$ if this is the case. We do this because it means we have found an uncongested vertex that makes more progress towards t and hence, we shouldn't return back to the previous ϕ. We now branch off into two cases:

(2.1) If $\lambda_{\text{start}} = \emptyset$ (see Figure 3.7), update $\lambda_{\text{start}} = w$, $\mathcal{I} = \mathcal{B}^{+}(|sw| - \lambda_{\text{dist}})$ and $\mathcal{R} = \mathcal{R} + (\mathcal{I} - \mathcal{I}')$. We note that despite w being an uncongested vertex, the path up to w would still be subject to the congestion factor, thus warranting the update of $\mathcal{I}$. This marks the beginning of a potentially uncongested section of the path in T_{st}.

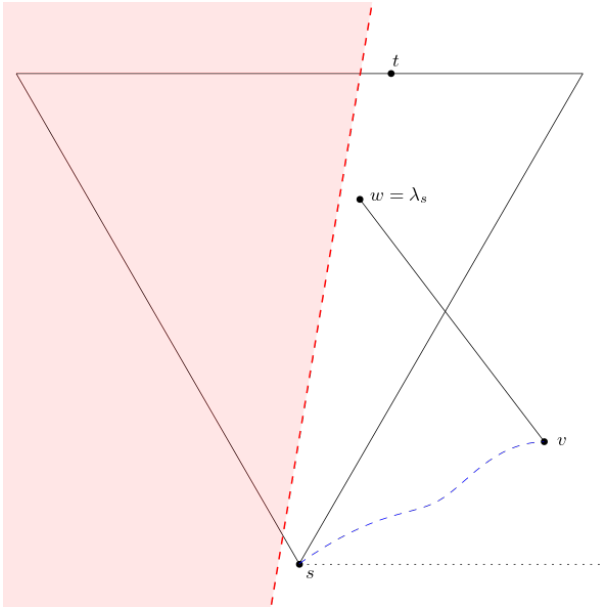
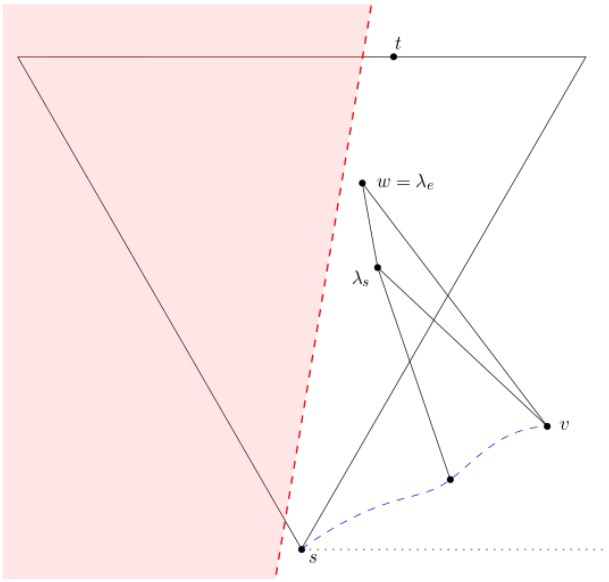


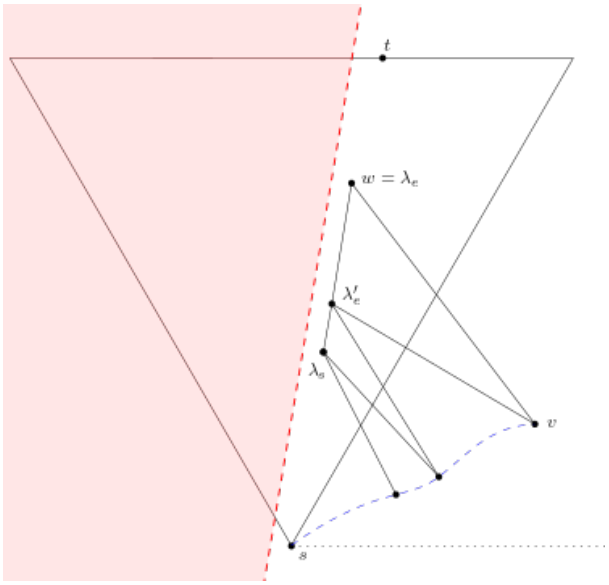
FIGURE 3.7. **Case 2.1**

(2.2) If $\lambda_{\text{start}} \neq \emptyset$, check if $\lambda_{\text{end}} = \emptyset$

If $\lambda_{\text{end}} = \emptyset$, update $\mathcal{R} = \mathcal{R} + |\lambda_{\text{start}} w|$. Otherwise, if $\lambda_{\text{end}} \neq \emptyset$, update $\mathcal{R} = \mathcal{R} - |\lambda_{\text{start}} \lambda_{\text{end}}| + |\lambda_{\text{start}} w|$. Finally, update $\lambda_{\text{end}} = w$. This corresponds to extending the bound on the uncongested section of the path in T_{st}.



(a) **Case 2.2**, where $\lambda_{\text{end}} = \emptyset$.



(b) **Case 2.2**, where $\lambda_{\text{end}} \neq \emptyset$. λ'_e represents the vertex which was assigned to λ_{end} before seeing w.

FIGURE 3.8. **Case 2.2**

(3) If v is connected to a mix of congested and uncongested vertices (see Figure 3.9), process each vertex from bottom to top by following the criteria outlined in **Case 1** and **Case 2**.

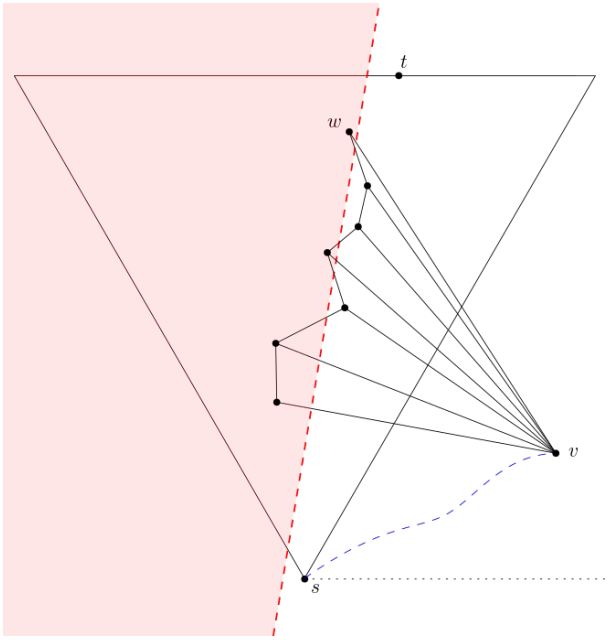


FIGURE 3.9. **Case 3**

(4) If v is connected to multiple uncongested vertices and no congested vertex in T_{st} (see Figure 3.10), process each vertex from bottom to top following **Case 2**. We note that afterwards, the highest vertex is set to λ_{end}.

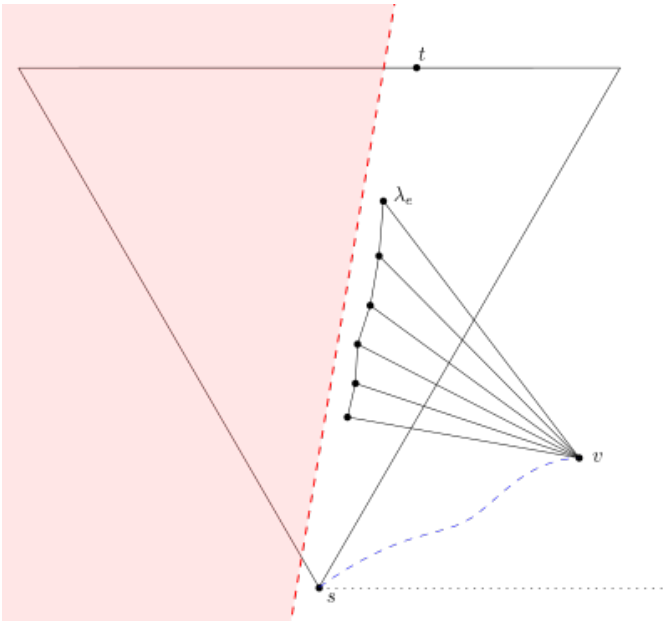


FIGURE 3.10. **Case 4**

(5) If v is connected to multiple congested vertices and no uncongested vertex in T_{st} (see Figure 3.11), select the highest vertex to be w and follow **Case 1**.

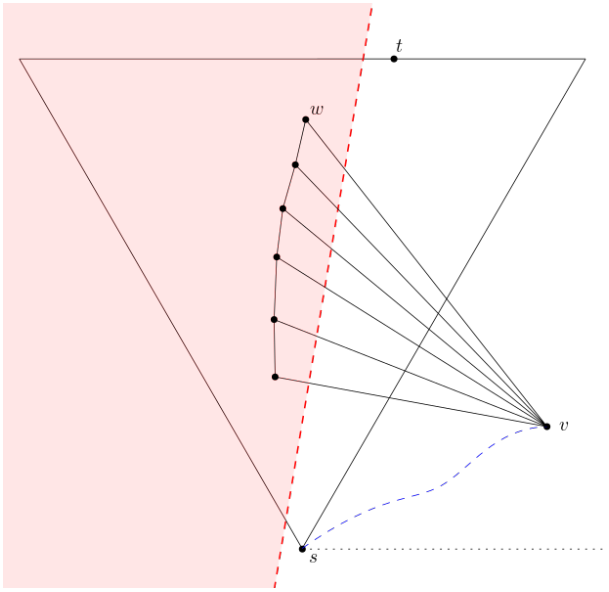


FIGURE 3.11. **Case 5**

We note that if we are searching backwards, we will not see any new vertices which can expand our exploration budget. Instead, we now update $\mathcal{R} = \min(\mathcal{R}, c|\phi t|)$. The latter parameter is a lower-bound on the path through the congested region at ϕ, the vertex identified as the most suitable to enter the congested region from. This adjustment tightens our exploration budget and stops our algorithm from exploring excessively when ϕ offers a reasonably short path through the congested region.

We now outline the case for updating our exploration budget if we re-enter T_{st}.

Canonical triangle budget update: Recall that we update our budget by considering vertices visible from outside T_{st}, which is equivalent to the path that we are currently following. As a result, when we continue to traverse upwards in T_{st} towards t, we do not change $\mathcal{R}$. However, we set λ_{start} to the first vertex in T_{st} we have traversed to and continue to update λ_{end} to the most recent vertex we have traversed to T_{ts}. This is to ensure that λ_{dist} is correctly maintained as it will be used if we have to leave T_{st}. Finally, if we see a congested vertex w above us that we cannot circumvent, we will update $\mathcal{I} = \mathcal{B}^{+}(|sw| - \lambda_{\text{dist}})$ and $\mathcal{R} = \mathcal{R} + (\mathcal{I} - \mathcal{I}')$.

3.2.2 Negative routing

Without loss of generality, let $t \in \bar{C}_0^s$. When we route positively outside of T_{st}, we explore in a positive cone of t. The opposite is true for routing negatively, which may require us to route to a vertex below t to avoid the congested region. We maintain a few additional variables:

- $\mathcal{R}_2$: assists in updating our exploration budget, initialised as $\min(|st|, \mathcal{R})$.
- $A_{\mathcal{B}}$: assists in updating our exploration budget, initialised as 0.
- $v', v'', A_1, A_2, A_{\text{return}}$: locations of specific vertices, initialised as $\emptyset$.

We define the exploration budget function for negative routing $\mathcal{B}^-(|su| - \lambda_{\text{dist}} - A_{\mathcal{B}}) = \frac{2c\sin(\alpha-\frac{\pi}{6})+c^2+1}{2(\sin(\alpha-\frac{\pi}{6})+c)} \cdot (|su| - \lambda_{\text{dist}} - A_{\mathcal{B}})$, where α is the angle st makes against the right boundary of C_0^t. $\mathcal{I}$ and $\mathcal{R}$ are both initially set as $\mathcal{B}^-(|su| - \lambda_{\text{dist}} - A_{\mathcal{B}})$.

We will expand on these variables and their use cases in the following sections.

3.2.2.1 Exploration path selection

At every step in our exploration path, we subtract from $\mathcal{R}$ and add the edge length to dist(ϕ). Regardless of if we are in C_2^t or $\bar{C}_1^t$, if there are no vertices left to explore or traversing to our selected vertex would cause $\mathcal{R} < 0$, we enter the return phase (described in Section 3.3).

When routing negatively, there could potentially be other uncongested vertices in T_{ts} to traverse to. Starting at s, we first consider any uncongested vertices in T_{ts}. If there are multiple uncongested vertices in T_{ts} to choose from, we choose the first vertex u anticlockwise from the right boundary of C_0^t.

If there are no vertices in T_{ts}, we will have to traverse outside of T_{ts}. We consider the first vertex u in an anticlockwise direction from the right boundary of T_{ts} that is outside of T_{ts}. There are two cases for where u is:

(1) If $u \in \bar{C}_0^v$ or C_1^v, proceed as normal.
(2) If $u \in C_2^v$, this means that there are either no vertices in $\bar{C}_0^v$ or if they do exist, they are blocked by vertices in T_{st}.

In the latter case, a configuration such as the one depicted in Figure 3.12 could exist. Evidently, it is unfavourable to stay close to the boundary of T_{ts} as this would result in repeatedly going back and forth

between the outermost path and the boundary of T_{ts}. This causes our algorithm to waste exploration budget while making little progress towards t. The intuition is to avoid regions where a configuration of vertices like that can occur and only enter it occasionally if it seems like there may be a path to t.

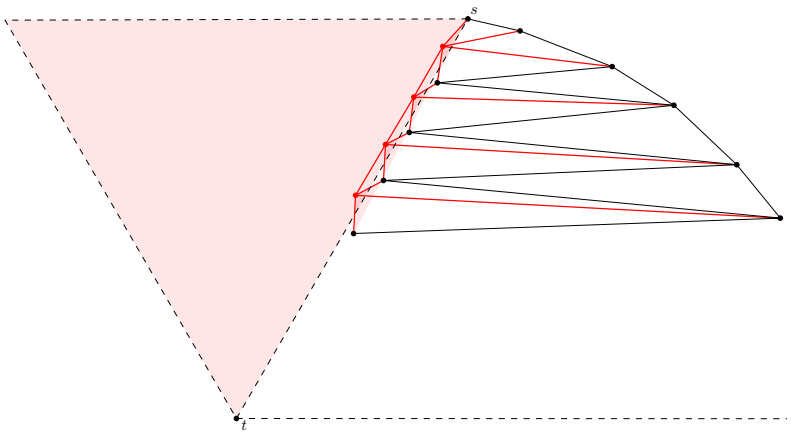


FIGURE 3.12. A configuration of points where choosing to stay close to the boundary of T_{ts} leads to a bad exploration path.

First, we save our current vertex as v'. Constructing a horizontal line through v, we compute its intersection with the right border of T_{ts}. This intersection will be saved under the variable v''. From v'', we create a line which forms a $\frac{\pi}{3}$ angle with the right boundary of T_{ts} and ends at the horizontal line through t. We will refer to the region bounded by $v''t$, the horizontal line through t and the line angled at $\frac{\pi}{3}$ through v'' as A. Region A potentially contains vertices outside of T_{ts} which are not part of a viable path to t and hence, we should be careful with any vertices we see in this area. An example is depicted in Figure 3.13.

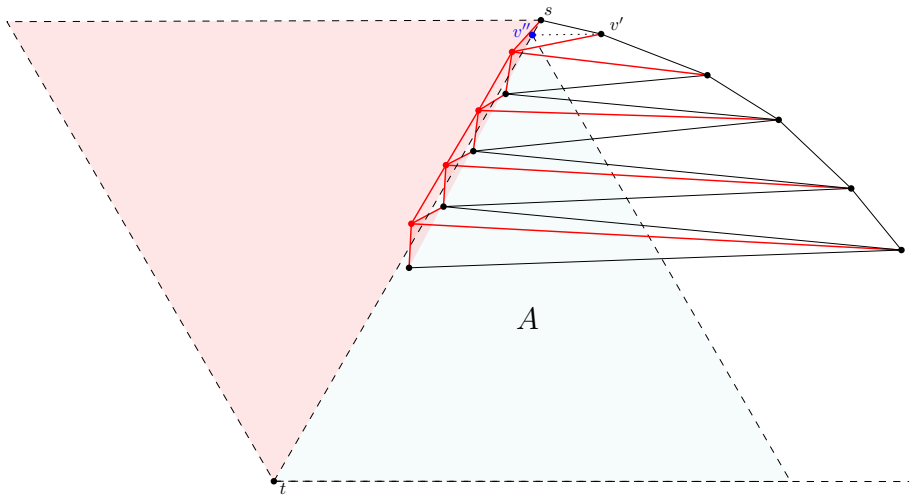


FIGURE 3.13. Construction of region A.

In our traversal, we use our secondary budget $\mathcal{R}_2$, which is initialised as $\mathcal{R}_2 = \min(\mathcal{R}, |st|)$. Taking the minimum ensures that $\mathcal{R}_2$ will always be less than our primary budget $\mathcal{R}$. If we can see vertices in A, we check if the the lowest vertex in $T_{ts} \cup A$ is uncongested. If that is the case, we subtract the distance we travel from $\mathcal{R}_2$. This is because we do not want our algorithm to traverse out arbitrarily far if there potentially exists a path to t which is close to T_{ts}. We had previously observed in Figure 3.12 that frequently checking in A can lead to arbitrarily poor exploration paths. As a result, our algorithm will only check in A if there are no vertices outside of $T_{ts} \cup A$ which can be traversed to without causing $\mathcal{R}_2$ to be less than 0.

If we run out of $\mathcal{R}_2$, we enter A at the lowest vertex we can see. The objective at this stage is to stay in $T_{ts} \cup A$ for as long as possible. During our traversal in $T_{ts} \cup A$, we also actively update the lowest visible uncongested vertex in T_{ts} to be ϕ as this represents the vertex that makes the most progress towards t. We also perform the ϕ-check at every vertex we traverse to in A. This is because vertices in A are considered reasonably close to T_{ts} and may offer a good position to enter the congested region at.

Let our current vertex be v. Once in A, we take vertices in C_1^v or $\bar{C}_0^v$ which are in A until there are no more to follow. If there are multiple vertices to choose from, take the first vertex in a clockwise direction from the right boundary of T_{ts}. These steps allow us to route along the edge of T_{ts} so we can gain information on the path in T_{ts} while moving towards t. Further details on how we update our budget when in A is covered in Section 3.2.2.2. Once there are no more vertices in A to traverse to, we check if there is an uncongested vertex in T_{ts} in C_1^v. If that is the case, we traverse in T_{ts} and repeatedly select

vertices in T_{ts} that are closest to the right boundary of T_{ts} in either $C_1^v, \bar{C}_0^v$ or C_2^v. If no uncongested vertices exist, we leave T_{ts} by taking the lowest uncongested vertex in either $\bar{C}_0^v$ or C_2^v. If we remain in A, we repeat the steps above to continue routing along the edge of T_{ts}.

If no uncongested vertices exist in $T_{ts} \cup A$, we should now leave A in case there is some uncongested path outside of A. There exist two cases now:

(1) We can see a vertex $w \in C_2^t$.
(2) We can not see a vertex $w \in C_2^t$.

Case 1: If we can see a vertex $w \in C_2^t$, we check if $|wt| \leq \text{dist}(\phi) + c|\phi t|$ and traverse to w if that is the case. This constraint ensures that we do not carelessly traverse to a vertex that may be arbitrarily far away in comparison to a path entering the congested region. At w, we can now enter the routing phase to t (described in Section 3.1). Otherwise, we enter the return phase (described in Section 3.3).

Case 2: In this case, we leave $T_{ts} \cup A$ by repeatedly taking vertices in C_2^v. If at any point we are connected to a vertex $w \in C_2^t$ in either C_2^v, we move to **Case 1**. If we reach a vertex outside of A in $\bar{C}_1^t$, we set our current vertex as A_{return} and repeat the steps outlined at the start of this section. We now set $\mathcal{R}_2 = \min(\mathcal{R}, 1.5|v'v|)$. This resembles an exponential search so that we do not frequently enter A and waste our exploration budget.

If we reach C_2^t, whether from some vertex in $T_{ts} \cup A$ or outside of it, we enter the routing phase to t (described in Section 3.1). Since t is to the right of the congested half-plane, C_2^t is guaranteed to be congestion-free.

If we are connected to t at any point in our exploration, we should traverse directly to it as that will conclude our search.

3.2.2.2 Budget update

Let v be the vertex we are currently at. The instructions for budget allocation outlined in Section 3.2.1.2 also apply to negative routing. One key difference is that we may be blocked by chains of vertices in region A instead and not be able to see any vertices in T_{ts} constantly. We split the budget update instructions into three cases:

- We are currently not in A and outside of T_{ts} or A does not exist

- We are currently in A
- We are currently in T_{ts}

Case 1: To update our budget, we consider vertices in $\bar{C}_2^v$ or C_1^v. As we traverse along the exterior of $T_{ts} \cup A$ we check vertices in either $\bar{C}_2^v$ or C_1^v. We use the variable u, introduced in Section 3.2 to keep track of which vertices we have already seen. This ensures that our algorithm will not wrongly adjust the exploration budget twice for the same vertex. At each step, we process vertices we can see in $\bar{C}_2^v$ and C_1^v from top to bottom. We follow the same instructions in Section 3.2.1.2 when dealing with vertices in T_{ts}.

We now define how to process vertices we see in A. We ensure that our exploration budget will never exceed the length of the path in T_{ts} by treating sections of the graph covered by vertices in region A as completely uncongested. To do so, we sum up the vertical distance of an uninterrupted chain of vertices in A. This is done by using the following two variables:

- A_1: tracks the first vertex in A that we have seen
- A_2: tracks the most recent vertex in A that we have seen.

As an example, in Figure 3.14, we are unable to see the path in T_{ts} at v. As a result, we assume that the purple region is completely uncongested to ensure we lower-bound the path in T_{ts}.

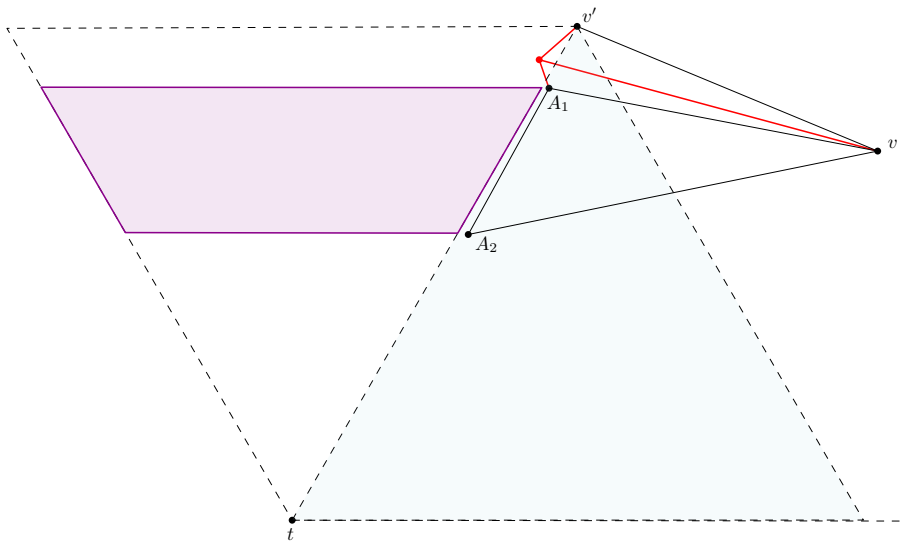


FIGURE 3.14. Example of consecutive vertices in A blocking our vision of the path in T_{ts}.

Let us assume we now see some vertex in A which is now set to A_1. Processing each vertex from top to bottom, we update A_2 to the lowest vertex in $T_{ts} \cup A$ which does not have some congested vertex in T_{ts} separating it. We then increment $A_{\mathcal{B}}$ by $|A_1A_2|$. Recall that our budget function is defined as $\mathcal{B}^-(|su| - \lambda_{\text{dist}} - A_{\mathcal{B}})$, where the parameter corresponds to the lower-bound of the congested path in T_{ts}. Subtracting $A_{\mathcal{B}}$ reflects us considering that specific portion of T_{ts} as uncongested. If some congested vertex in T_{ts} can be seen, we reset A_1 and A_2 to $\emptyset$ as that indicates we are not being continuously blocked by vertices in A. In Figure 3.15, assume we are at v. We process u first and set it as A_1 as it could indicate the start of a chain. However, we see a vertex $x \in T_{ts}$, which implies that there is no path in A. This causes us to reset A_1 and set $A_1 = w$ when we process it next. It is worth noting that if x was uncongested, we would retain $A_1 = u$ and set $A_2 = w$.

In addition, if we see vertices in T_{ts} which are uncongested while $A_1 \neq \emptyset$, we should not update any of our λ variables. Recall that λ_{dist} corresponds to the length of an uncongested path in T_{ts}. This uncongested section is already being accounted for with $A_{\mathcal{B}}$ so we do not need to keep track of it again.

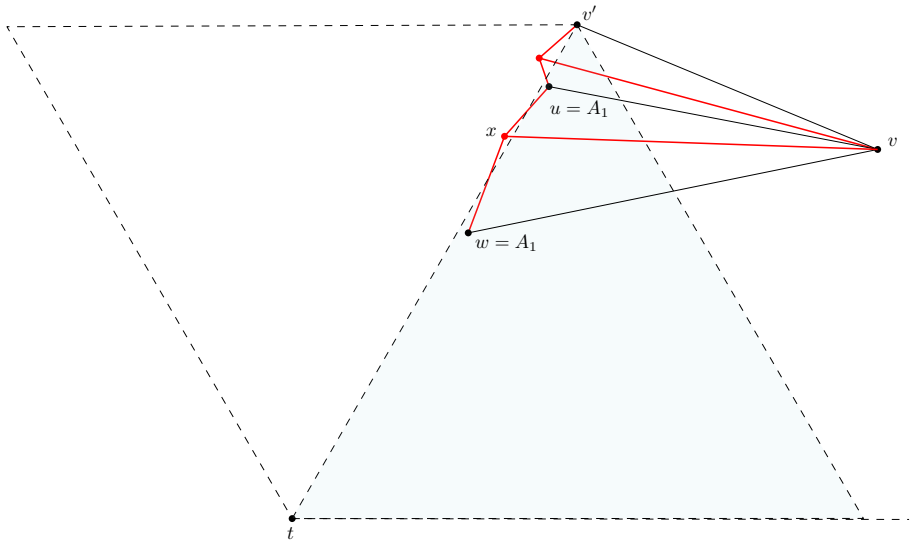


FIGURE 3.15. Example of vertices in A being separated by a vertex in T_{ts}.

Likewise, if we are unable to see vertices in T_{ts}, we should also use vertices in A to extend $\mathcal{R}$. We do so by incrementing $\mathcal{R}$ by $|A_1A_2|$. When a new vertex w should be updated to be A_2 and $A_2 \neq \emptyset$, we increment $\mathcal{R}$ and $A_{\mathcal{B}}$ by $|A_1w| - |A_1A_2|$.

Case 2: If we are in A, we only observe vertices in C_1^v to update our exploration budget. This is because we should have already processed everything above the current vertex we have entered A at. Once we

can see vertices in T_{ts}, the same budget reallocation steps as in Section 3.2.1.2 apply. As we are using a positive cone, C_1^v, to observe the path within T_{ts}, we can see a maximum of one vertex at a time so **Case 3, Case 4** and **Case 5** in Section 3.2.1.2 cannot occur.

Case 3: If we are in T_{ts}, our algorithm will follow vertices along the right boundary of T_{ts}. We follow the same steps outlined in the **Canonical triangle budget update** case in Section 3.2.1.2, with the difference of updating $\mathcal{I} = \mathcal{B}^-(|sw| - \lambda_{\text{dist}} - A_{\mathcal{B}})$.

3.3 Return Phase

During this phase, our algorithm will backtrack to ϕ. This phase is entered when there are no more uncongested vertices to explore or if we have exhausted the exploration budget. At each vertex we progress to, we check if we are connected to ϕ and proceed to it if found. We enter the congested routing phase (described in Section 3.1.1) at ϕ. Let v be our current vertex. The following two sections will describe how to backtrack if we were routing positively or negatively before entering the return phase.

3.3.1 Returning after positively routing

There are three possible cases when we are routing positively:

(1) We are in $\bar{C}_1^s$ and BACKWARDS is False
(2) We are in $\bar{C}_1^s$ or C_2^s and BACKWARDS is True
(3) We are in C_2^t

We note that the first two cases are in $\bar{C}_0^t$.

Case 1: This is the simplest case as this means we have only been taking edges which are close to T_{st}. Hence, we can backtrack by selecting vertices in either $\bar{C}_0^v$ and C_1^v which are closest to the right boundary of T_{st}.

Case 2: Since BACKWARDS is True, that indicates we have begun searching in the opposite direction of t. Recall that when searching in the opposite direction, we only take vertices in the C_2 cone (see Section 3.2). Thus, we can return by considering vertices in $\bar{C}_2^v$. If there are multiple vertices to choose from, we take the first vertex clockwise from the the right boundary of $\bar{C}_0^t$. If there are no uncongested vertices

to further traverse to, that means we have reached the vertex where we had set BACKWARDS to be True. We are either at ϕ or ϕ is below us so we now enter **Case 1**.

Case 3: Being in this case would imply that BACKWARDS is False since otherwise, we would have entered the routing phase when we reached C_2^t. To return, we select the first vertex anticlockwise from the right border of $\bar{C}_0^t$ in either $\bar{C}_0^v, C_2^v$ or $\bar{C}_1^v$. This is done until we can see a vertex in $\bar{C}_0^t$. If there are multiple vertices to choose from, we choose the closest vertex to the right border of T_{ts} and enter **Case 1**.

3.3.2 Returning after negatively routing

We split routing negatively into the following cases:

(1) Region A exists
(2) Region A does not exist

Case 1: If A exists, we may potentially have to enter A to reach ϕ. We backtrack by selecting vertices closest to the boundary of $T_{ts} \cup A$ until we reach A_{return}. This variable indicates the vertex which we last exited A at and thus, we now enter A. Since we set ϕ to be the lowest uncongested vertex in A, we should repeatedly select the lowest vertex in $\bar{C}_2^u$ each time to get to ϕ. The same steps apply if we enter the return phase while already in A.

Case 2: If region A does not exist, we can simply take vertices which are closest to the right boundary of T_{ts} in $\bar{C}_2^u, C_0^u$ or $\bar{C}_u^1$ until we eventually are connected to ϕ.

CHAPTER 4

Half-plane routing analysis

4.1 Budget allocation

In Chapter 3, we introduced a positive routing budget function $\mathcal{B}^+$ and a negative routing budget function $\mathcal{B}^-$ which both take some Euclidean distance as a parameter. This distance represents a lower-bound on the length of the congested path up to the lowest vertex we can see in the canonical triangle of s and t.

Recall that our congestion-aware routing algorithm estimates this lower-bound by using the Euclidean distance between the last vertex outside of the congested region before entering the search phase and a congested vertex that our exploration has revealed an edge to. If multiple congested vertices are seen, the exploration budget will be based on the furthest vertex in the direction of t.

Bose et al.'s algorithm has been proven to be an optimal local routing algorithm for the half-Θ_6-graph. Hence, our algorithm uses the path taken by Bose et al.'s algorithm to lower-bound the length of the path in the canonical triangle of s and t. In this chapter, we will refer to the *optimal path* as the one taken by Bose et al.'s algorithm. We define the set O as the set of outermost vertices in the congested region which are connected to any vertex outside of the congested region. When our algorithm is in the search phase, we use vertices in O that we have seen to allocate exploration budget. In other words, our algorithm is using vertices in O to gauge the length of the optimal path in the canonical triangle. However, the optimal path will not always follow the outermost vertices in the congested region. Hence, we may be "blocked" by vertices in O from seeing the actual optimal path through the congested region.

As an example, consider Figure 4.1. The optimal path is outlined in green while our algorithm's search for an alternative path is outlined in blue. At s, our initial exploration budget is based on $|su|$, which takes us to x. However, at x, we are blocked by w from seeing v. This results in the exploration budget being based on a vertex not on the optimal path.

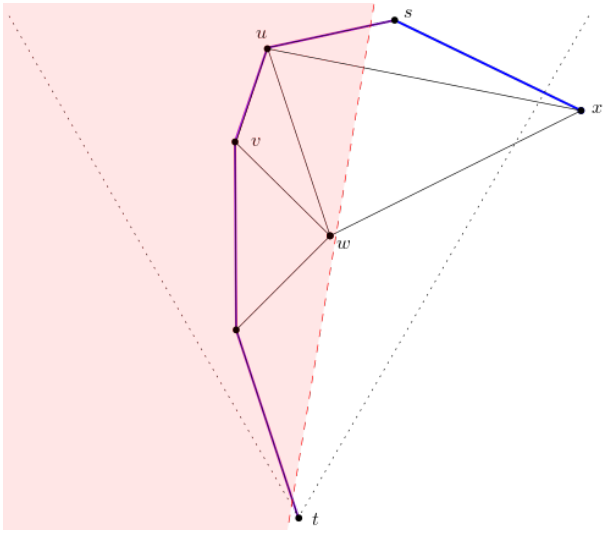


FIGURE 4.1. Example of being unable to see the optimal path. Red shaded region represents the congested region. Purple edges represent the optimal path and blue represents edges taken during our algorithm's exploration.

We argue that in instances where our algorithm cannot see the optimal path, the exploration budget allocated will never exceed the length of the optimal path. Let s be the last vertex along the optimal path outside of the congested region and u be a congested vertex connected to s, which is also on the optimal path in the canonical triangle s and t. Vertex u is connected to two vertices, v and w, which are both in the canonical triangle of s and t. Let v be the vertex Bose et al.'s algorithm would choose over w and let w be the furthest congested vertex in O which we can see. If w was on the optimal path, we can lower-bound the optimal path length with $|sw|$ and charge our allocated exploration budget to the length of the optimal path. Thus, we assume that w is not on the optimal path and $|uv| < |uw|$. As our algorithm can only see w, the exploration budget will now be based on $|sw|$.

We argue that the optimal path must include a vertex z such that the path up to z (which may or may not include z itself) is in the congested region and $|sz| > |sw|$. If z is not in the congested region, it must be connected to a congested vertex on the optimal path. This ensures that all the edges up to z are subject to the multiplicative congestion factor. By proving the existence of z, we can lower-bound the length of the optimal path through the congested region with $|sz|$ and charge the allocated exploration budget of $|sw|$ to it. In the following proof, whenever a cone is mentioned, we are referring to the region created by the intersection of the cone with the canonical triangle of s and t.

4.1.1 Negatively routing from s

Without loss of generality, let $t \in \bar{C}_0^s$. We set to show that z will always exist in both of the following cases:

(1) s is to the right or directly above w.

(2) s is to the left of w.

To identify the potential locations of w, we introduce the following lemma.

LEMMA 4. *Vertices v and w must be in the same cone of u.*

PROOF. For the sake of contradiction, let us assume that v and w are in different cones of u. As the optimal routing algorithm favours staying close to the empty side of T_{tu}, we consider an instance where $v \in \bar{C}_0^u$ and $w \in C_2^u$. The same argument applies if $v \in C_1^u$ and $w \in C_2^u$. Since w is in the congested region and s is not in the congested region, $s \in \bar{C}_1^u$. This leaves w to be in 2 possible cones: $\bar{C}_0^s$ or C_1^s.

If $w \in \bar{C}_0^s$, the optimal routing algorithm would select w instead of u at s as it is in the same cone as t (see Figure 4.2). This contradicts our original assumption that w is not on the optimal path. Otherwise, if $w \in C_1^s$, w would be contained in T_{su}, which makes it closer to s than u is. Since C_1^s is a positive cone of s, there would be an edge from s to w instead of u (see Figure 4.3). Thus, by contradiction, v and w must be in the same cone of u.

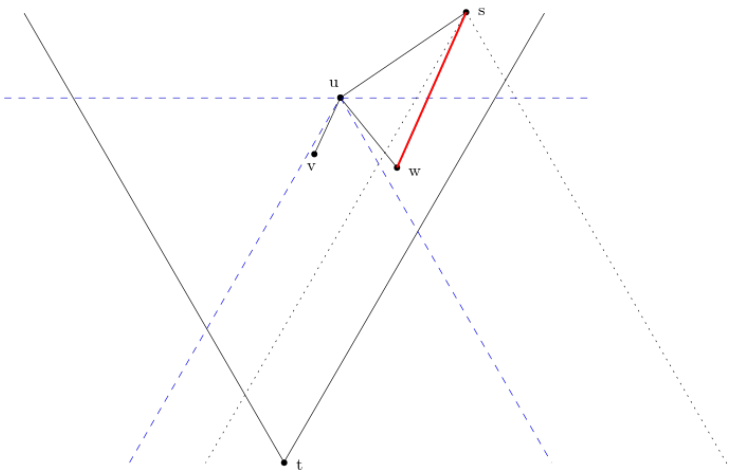


FIGURE 4.2. The case $w \in \bar{C}_0^s$. The red line indicates the edge the optimal routing algorithm would have selected.

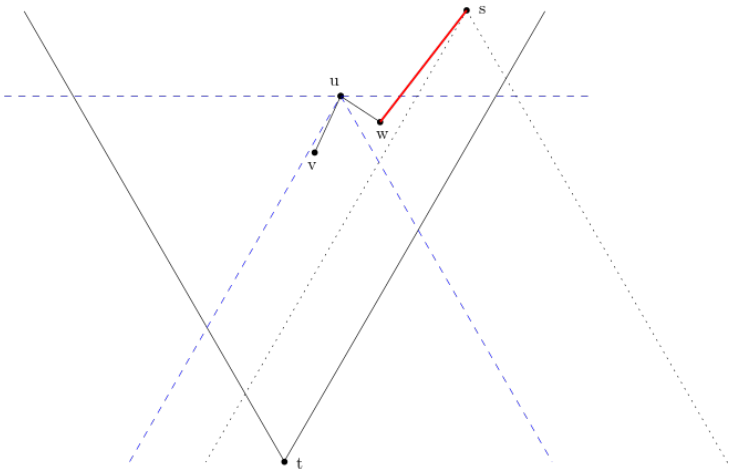


FIGURE 4.3. The case $w \in C_1^s$

We remark that the case when $v \in C_1^u$ and $w \in \bar{C}_0^u$ cannot exist. This is because Bose et al.'s algorithm prioritises following vertices in the $\bar{C}_0$ cone, placing w on the optimal path and contradicting our original assumption. □

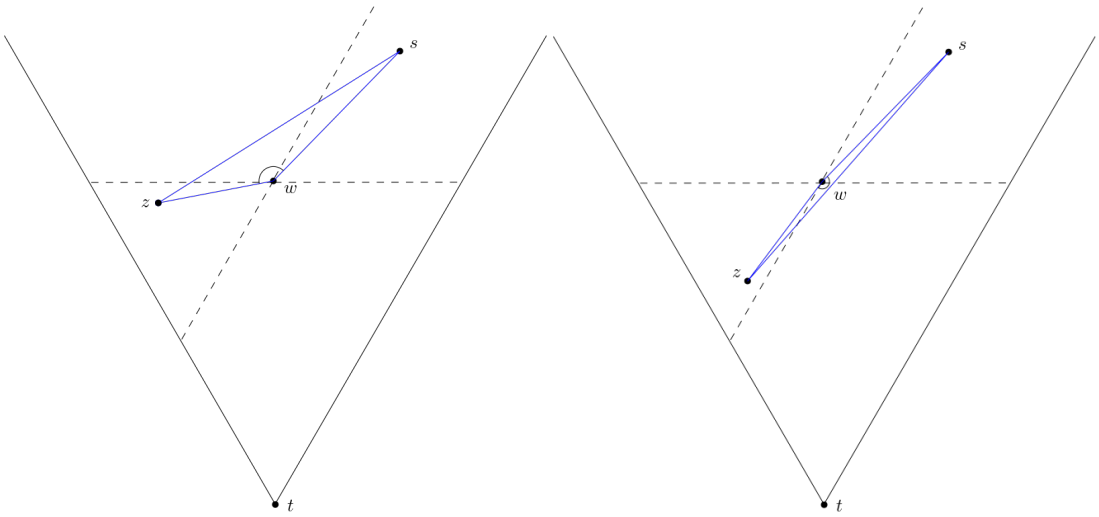
We now continue to prove that the optimal path must include a vertex z such that the path up to z is in the congested region and $|sz| > |sw|$. This proof makes extensive use of the following property:

OBSERVATION 1. *Within any triangle, the largest angle must be opposite the longest side.*

We also introduce a useful lemma for our proof:

LEMMA 5. *For any pair of vertices, s and z, that lie in opposite cones of w, $|sz| > |sw|$.*

PROOF. Without loss of generality, let us assume $s \in \bar{C}_1^w$ and $z \in C_1^w$. As two cones separate $\bar{C}_1^w$ and C_1^w, $\angle swz$ must be at least $\frac{2\pi}{3}$. Depending on the position of z with respect to s, $\angle swz$ can constructed in either a clockwise or anticlockwise direction from sw (see Figure 4.4). We take whichever direction yields $\angle szw < \pi$ as it ensures $\triangle szw$ remains a valid triangle. By Observation 1, this makes $|sz|$ the largest side of $\triangle szw$. In the case where s, w and z are collinear, sz would be an extension of sw, making it trivially true that $|sz| > |sw|$. Therefore, if a vertex z lies in the cone of w which is opposite to the cone that contains s, $|sz| > |sw|$.



(a) Anticlockwise from sw (b) Clockwise from sw

FIGURE 4.4. Construction of $\angle szw$ (negative routing)

□

4.1.1.1 Case 1

We now cover the case where s is to the right or directly above w. As a result of Lemma 4, v and w must both be in a negative cone of u. Without loss of generality, let v and w be in $\bar{C}_0^u$ and let w be to the right of v. As s is assumed to be to the right of w or above w, u or is either in $\bar{C}_0^s$ or C_1^s. We can now identify three subcases:

CASE 1.1: v and w are both in $\bar{C}_0^s$.
CASE 1.2: $v \in C_1^s$ and $w \in \bar{C}_0^s$.
CASE 1.3: v and w are both in C_1^s.

For each of the subcases, we argue that the optimal routing algorithm must traverse to a vertex z such that $|sz| > |sw|$. Under the general position assumption, we note that collinearity between s, w and z can only occur in **Case 1.3**.

Case 1.1: Let z be a vertex on the optimal path which is either inside the congested region or connected to a vertex in O. The half-plane defining the congested region separates w and s. Since w is within the congested region, at least part of C_1^w must be within the congested region. For a path to go from v to t without going through w, there must be a vertex which lies in C_1^w (see Figure 4.5). This is because any

vertex in $\bar{C}_0^w$ will connect to w and not any vertex above w if C_1^w is empty. In addition, the half-Θ_6-graph is planar and uw separates v from any potential vertices to the right of uw. Thus, there must be at least one vertex on the optimal path in C_1^w. If there exist multiple vertices on the optimal path in C_1^w, let z be the highest vertex amongst them. This ensures that the path up to z is in the congested region as there are no guarantees that the entirety of C_1^w is within the congested region. **Case 1.2**: The argument

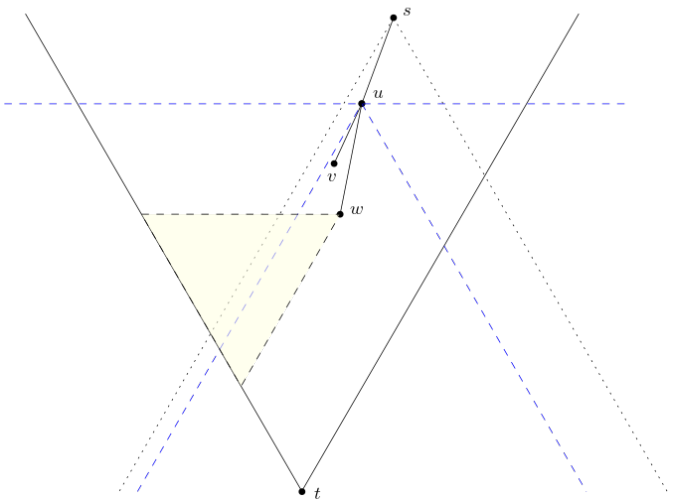


FIGURE 4.5. Potential locations for z in **Case 1.1** (negative routing)

for **Case 1.2** is analogous to **Case 1.1**. There must also exist a z on the optimal path in C_1^w (see Figure 4.6).

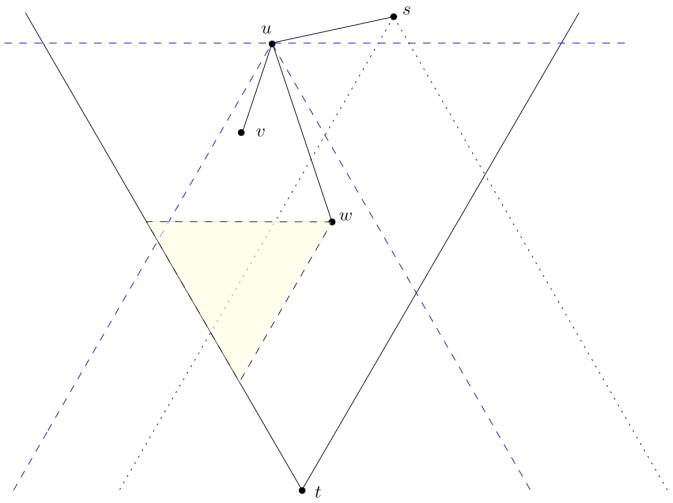


FIGURE 4.6. Potential locations for z in **Case 1.2** (negative routing)

We now argue that any point z in C_1^w will result in $|sz| > |sw|$, given s is to the right or directly above w. Let us consider the triangle $\triangle szw$. To minimise $|sz|, z$ would lie arbitrarily close to the dashed border dividing C_1^w and $\bar{C}_2^w$ (see Figure 4.7 and Figure 4.8). As $s \in C_0^w$, $\bar{C}_2^w$ separates s and z. Let α be the angle that s makes with the left cone boundary of C_0^w. We can define $\angle swz \geq \frac{\pi}{3} + \alpha$. As we have assumed that s is to the right or directly above w, $\alpha \geq \frac{\pi}{6}$. This results in $\angle swz \geq \frac{\pi}{2}$. Using Observation 1, we can conclude that $|sz|$, which is opposite $\angle swz$, is the largest side in $\triangle szw$ and hence, any $z \in C_1^w$ will imply $|sz| > |sw|$.

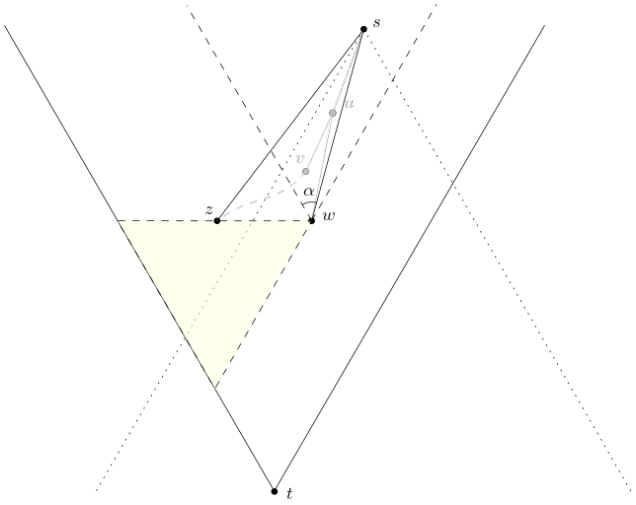


FIGURE 4.7. Minimising $\angle szw$ in **Case 1.1** (negative routing)

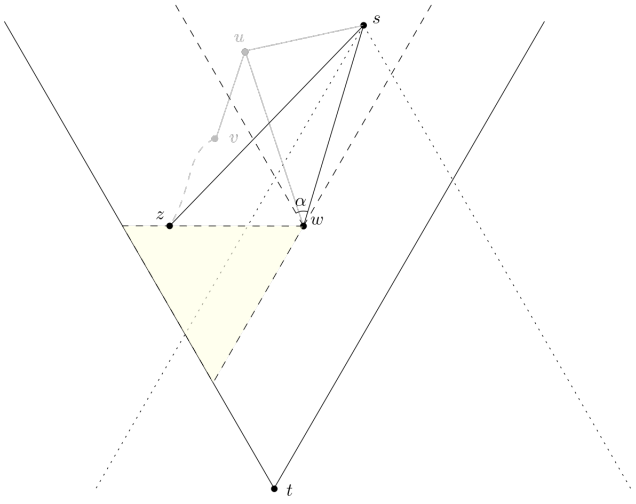


FIGURE 4.8. Minimising $\angle szw$ in **Case 1.2** (negative routing)

As a result, there must exist a vertex z on the optimal path, such that $|sz| > |sw|$ in **Case 1.1** and **Case 1.2**. Moreover, we can observe that our proof does not make use of the position of v and hence, v does not impact the potential locations of z. Thus, we can group future subcases where only the position of v differentiates them.

Case 1.3: A similar argument can be made for **Case 1.3**, in which there must be a $z \in C_1^w$ to ensure that the optimal path does not include w (see Figure 4.9).

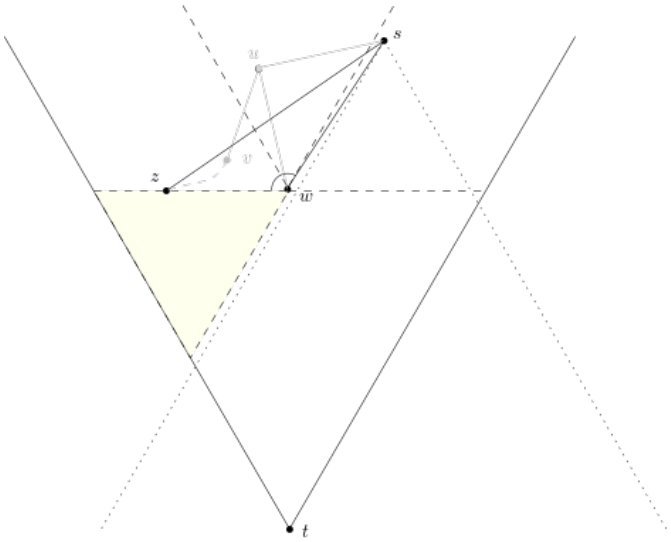


FIGURE 4.9. Potential locations for z in **Case 1.3** (negative routing)

We first observe that $s \in \bar{C}_1^w$ and $z \in C_1^w$. With s and z being in cones of w that are directly opposite one another, we can apply Lemma 5. Thus, there must exist a z on the optimal path through the congested region, such that $|sz| > |sw|$.

4.1.1.2 Case 2

If s is to the left of w, w must be in $\bar{C}_0^s$. We argue that w cannot be in C_2^s. Vertex v being positioned to the left of w while also being in the congested region suggests that the congested region is to the left of the line defining it. If $w \in C_2^s$, there does not exist a half-plane separating s and t from the congested region, contradicting that s and t are not in the congested region. As a result, w must be in $\bar{C}_0^s$ and only the first two subcases from **Case 1** apply:

CASE 2.1: v and w are both in $\bar{C}_0^s$.

CASE 2.2: $v \in C_1^s$ and $w \in \bar{C}_0^s$.

Analogous to **Case 1**, v's position is not relevant to the proof and we can make the same argument that holds for both subcases. For visualisation purposes, we will use **Case 2.1**. Let z be the first vertex on the optimal path which is outside of the congested region. As s is to the left of w and outside of the congested region, the half-plane must have a negative slope. This implies that C_1^w will be entirely contained in the congested region and hence, z can only lie in $\bar{C}_0^w$ or C_2^w (see Figure 4.10).

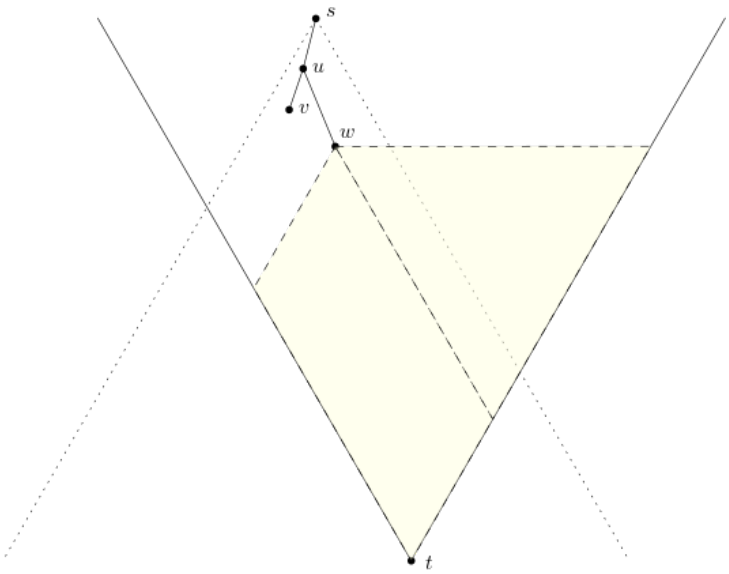


FIGURE 4.10. Potential locations for z in **Case 2** (negative routing)

If $z \in \bar{C}_0^w$, we can apply Lemma 5 as s and z are in opposite cones and conclude that $|sz| > |sw|$. Otherwise, if $z \in C_2^w$, we follow a symmetrical argument to the one made for **Case 1.12**. Consider $\angle szw$ clockwise from sw. Let α be the angle that s makes with the right boundary of C_0^w (see Figure 4.11). Therefore, we can express $\angle swz = \frac{\pi}{3} + \alpha$. As we have assumed that s is to the left of w, $\alpha > \frac{\pi}{6}$. This results in $\angle swz > \frac{\pi}{2}$, making it the largest angle in $\triangle szw$. Therefore, we can also conclude that $|sz| > |sw|$.

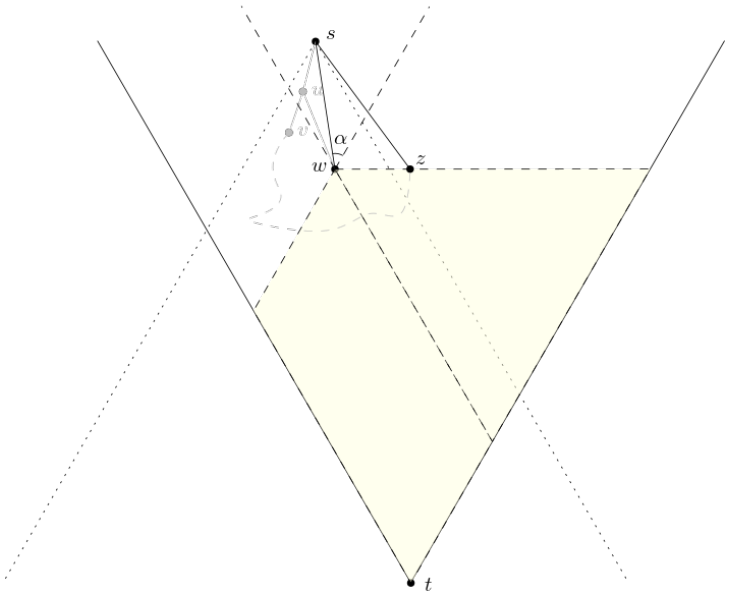


FIGURE 4.11. Minimising $\angle szw$ in **Case 2** (negative routing)

4.1.2 Positively routing from s

Without loss of generality, let $t \in C_0^s$. Using the same setup of vertices, u, v and w, as described in Section 4.1, we can now consider two cases:

(1) w is in a negative cone of t.
(2) w is in a positive cone of t.

Let z be either the first vertex on the optimal path after leaving the congested region or an arbitrary congested vertex on the optimal path. The crux of the argument for both cases is that z must lie in C_0^w. Since $s \in \bar{C}_0^w$, we can use Lemma 5 to conclude that $|sz| > |sw|$. This argument will be further expanded on in the following sections.

4.1.2.1 Case 1

If w is in a negative cone of t, $w \in \bar{C}_0^t$. This leaves two subcases for where v is with respect to t:

CASE 1.1: $v \in \bar{C}_0^t$.
CASE 1.2: $v \in C_1^t$.

The same argument holds regardless of the position of v. For visualisation purposes, we will use **Case 1.1**. As the half-plane defining the congested region separates w and t, it must intersect C_0^w, causing a section of C_0^w and everything to the left of it to be in the congested region. As $\bar{C}_2^w$ is to the left of C_0^w, $\bar{C}_2^w$ must be entirely in the congested region. Thus, to leave the congested region, the optimal path must have at least one vertex in C_0^w (see Figure 4.12).

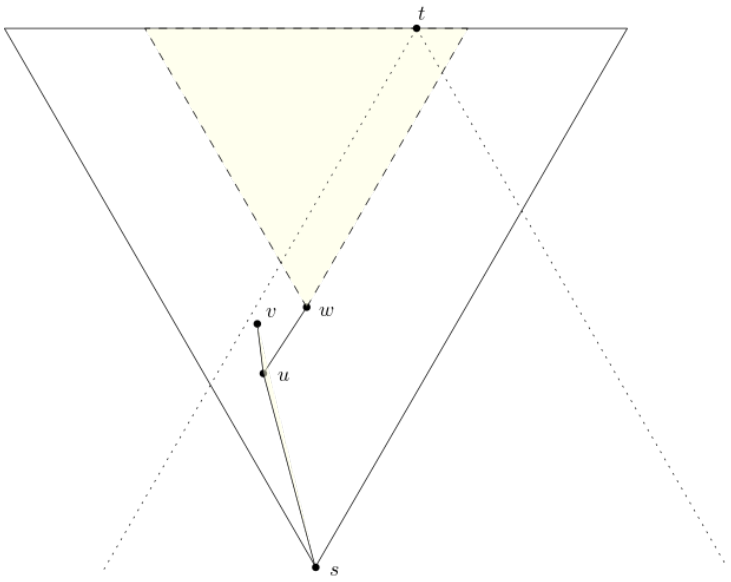


FIGURE 4.12. Potential locations for z in **Case 1** (positive routing)

If there exist multiple vertices on the optimal path in C_0^w, select the leftmost vertex as z to ensure that the path up to z is congested. An example is illustrated in Figure 4.13, in which the leftmost vertex in C_0^w is selected as z. This makes z the first vertex on the optimal path outside of the congested region. As z is connected to a vertex in A, the path from s to z is guaranteed to be entirely congested.

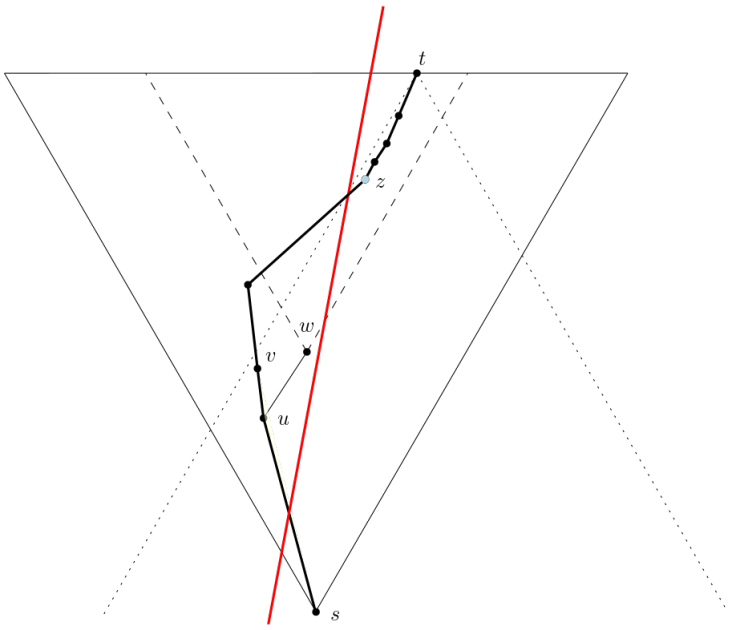


FIGURE 4.13. Selecting the leftmost vertex in C_0^w as z, highlighted in blue. Optimal path is outlined in bold; red line denotes the half-plane.

4.1.2.2 Case 2

Without loss of generality, let $w \in C_1^t$. If $w \in C_1^t$, then $v \in C_1^t$ since v is assumed to be to the left of w. There must be a vertex z in C_0^w on the optimal path for it to not include w (see Figure 4.14). This is because there cannot exist an edge connecting a vertex $a \in \bar{C}_1^w$ to a vertex in $\bar{C}_2^w$ as w would have a shorter projection onto the bisector of C_1^a. Analogous to **Case 1**, we select the leftmost vertex to be z if there are multiple vertices on the optimal path in C_0^w, completing this case.

This concludes our proof that using the straight line distance from s to any visible congested vertex will not overestimate the length of the optimal path.

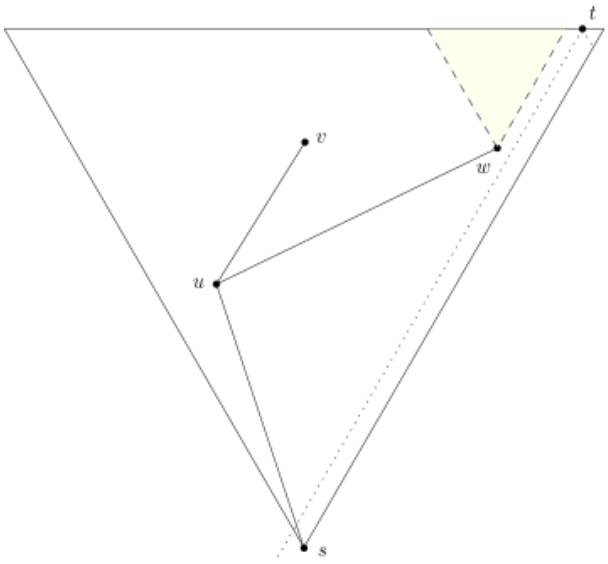


FIGURE 4.14. Potential locations for z in **Case 2** (positive routing)

4.1.3 Subtracting λ_{dist}

We continue to argue that we will never overestimate the length of the optimal path when $\lambda_{\text{dist}} \neq 0$. We observe that uncongested vertices can exist in the canonical triangle of s and t, which leads to potential uncongested sections of the path. The parameter in both $\mathcal{B}^+$ and $\mathcal{B}^-$ involves subtracting λ_{dist} from the Euclidean distance of s to the lowest visible congested vertex in the canonical triangle of s and t. We first make the following observation:

OBSERVATION 2. *The length of the projection of a line segment is at most the length of the line segment itself.*

Recall that λ_{dist} is the sum of uncongested path segments. As a result of Observation 2, directly subtracting λ_{dist} will continue to maintain a lower-bound on the length of the congested path. For example, consider Figure 4.15. Since u is the highest congested vertex visible from v, a vertex on our exploration path, $|su|$ will be a parameter in the exploration budget function. On our exploration path, we have seen two uncongested vertices, λ_s and λ_e, in the canonical triangle T_{st}. If we assume the optimal path traverses to these vertices before going from λ_e to u, we technically only need to subtract the perpendicular projection of $\lambda_s\lambda_e$ onto su. This is shown in blue in Figure 4.15. Since we directly subtract $\lambda_{\text{dist}} = |\lambda_s\lambda_e|$, we will be subtracting more than necessary but doing so will safely ensure that we never

overestimate the length of a congested path. Using Observation 2 again, we can also see that the distance $|\lambda_e u|$ will exceed $|\lambda'_e u|$, where λ'_e is the point on su when we project λ_e onto it. This reinforces that using $|su|$ as a lower-bound for the congested path is correct. The same argument can be applied

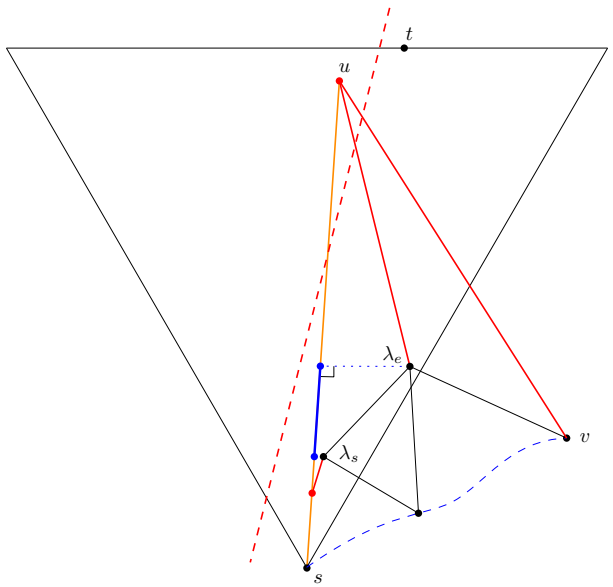


FIGURE 4.15. Example of how subtracting λ_{dist} is a lower-bound. The blue line segment represents the projection of $\lambda_s\lambda_e$ onto su.

for subtracting $A_{\mathcal{B}}$ when we route negatively. This concludes our proof that our algorithm will never overestimate the length of the optimal path.

4.2 Approximation ratio

To evaluate the effectiveness of our algorithm, we compute the approximation ratio of our algorithm's path length against the shortest path length. We make the following claim:

THEOREM 1. *The half-plane routing algorithm produces a 4-approximation of the shortest path length when routing positively and a* $((2\sqrt{3}+5)/\sqrt{3}) \approx 4.9$*-approximation when routing negatively.*

PROOF. Let s be the vertex we start from. Any distance we travel using Bose et al.'s algorithm in the routing phase would improve our approximation ratio so let us assume we immediately see a congested vertex we would normally traverse to. We consider the following four cases:

(1) There is no path around the congested region and the shortest path is directly through the congested region in the canonical triangle of s and t.
(2) Our algorithm runs out of exploration budget while the shortest path can be found if we traversed ϵ further than the allocated exploration budget.
(3) Our algorithm returns back to ϕ while the shortest path enters the congested region at some vertex v on the exploration path.
(4) Our algorithm finds an alternative path outside of the congested region after using up the entire exploration budget.

Let x denote our exploration budget. We begin by observing that x is based on the location of the furthest congested vertex in the direction of t. Thus, to upper-bound the distance our algorithm will traverse, we place x arbitrarily close to t. This also allows us to lower-bound the path in the canonical triangle of s and t with $c|st|$. To maximise the path length taken in the return phase (described in Section 3.3), we also ensure that $\phi = s$ for the entire duration of the search phase (described in Section 3.2).

In **Case 1**, **Case 2** and **Case 3**, our algorithm will expend x to explore out and x to return back to s. This results in a path length of $2x$, plus the congested path length in the canonical triangle of s and t. Thus, our analysis will centre around lower-bounding the shortest path length to determine which case will produce the largest approximation ratio. In our analysis, we will use the following property:

LEMMA 6. *In $\triangle abc$, $c^2 = a^2 + b^2 - 2ab \cdot \cos(\gamma)$, where γ is the angle opposite side c.*

We begin by concluding that **Case 4** will not contribute to the approximation ratio as it has a smaller approximation ratio when compared to **Case 2**. This is because not having to return to s will greatly reduce the length of the path produced by our algorithm, which reduces the approximation ratio. Further details for this specific case are analysed in Section 4.3.

4.2.1 Positive routing

Without loss of generality, let us assume $t \in C_0^s$. We begin by proving the first claim of Theorem 1:

LEMMA 7. *As $c \to \infty$, the half-plane routing algorithm produces a 4-approximation of the length of the shortest path when routing positively.*

PROOF. When routing positively from s, the path in the canonical triangle T_{st} is upper-bounded by $2c|st|$ as a result of Lemma 1. The lower-bound on the shortest path is $c|st|$. Let $|st|$ be the unit of length. From **Case 1**, we obtain the following approximation ratio:

$$\frac{2x + 2c}{c} \tag{4.1}$$

In **Case 2**, we consider the scenario where we run out of budget, traversing a distance of x towards t in $\bar{C}_1^s$. Let us assume there exists a single vertex $x_0 \in C_2^t$ which has an Euclidean distance of x away from s. As a result, we immediately run out of budget traversing to x_0. To lower-bound the length of the shortest path, let us assume that if we had traversed an arbitrarily small distance of ϵ further, there exists a direct edge to t. The length of the shortest path can now be expressed as $x + |x_0t|$.

To minimise $|x_0t|$, x_0 should be close to the horizontal line through t (see Figure 4.16). Consider $\angle stx_0$. We can express $\angle stx_0$ as $\frac{2\pi}{3} - \alpha$, where α is the angle st makes against the right boundary of C_0^s. Using Lemma 6 on $\triangle stx_0$, we can express:

$$x = \sqrt{|x_0t|^2 + 1 - 2|x_0t| \cos(\frac{2\pi}{3} - \alpha)}$$

This can be solved for a closed-form solution of $|x_0t|$:

$$|x_0t| = \frac{1}{2}\left(2\sin\left(\alpha - \frac{\pi}{6}\right) - \sqrt{2}\sqrt{2x^2 - \sin\left(2\alpha + \frac{\pi}{6}\right) - 1}\right)$$

We now obtain the approximation ratio for **Case 2**:

$$\frac{2x + 2c}{x + \frac{1}{2}\left(2\sin\left(\alpha - \frac{\pi}{6}\right) - \sqrt{2}\sqrt{2x^2 - \sin\left(2\alpha + \frac{\pi}{6}\right) - 1}\right)} \tag{4.2}$$

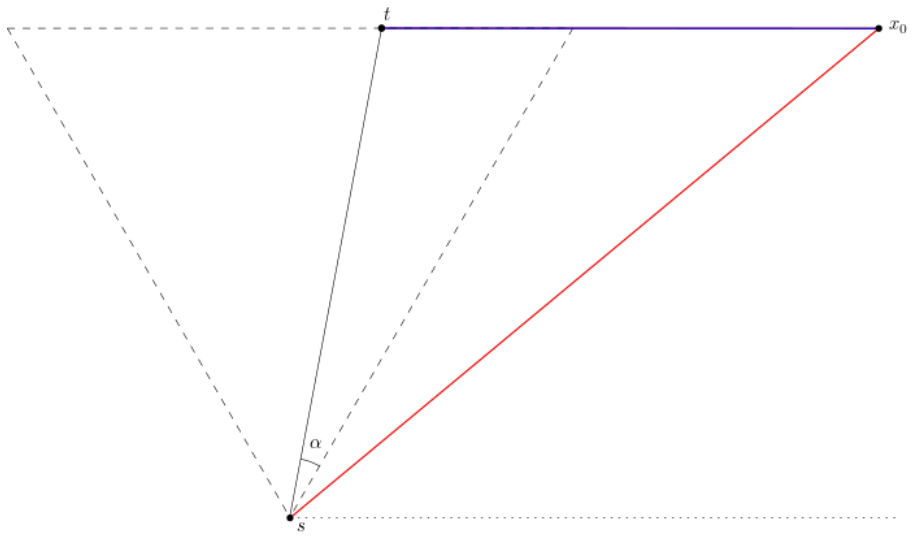


FIGURE 4.16. **Case 2** in positive routing. The red edge is our exploration, x, and the blue is $|x_0t|$.

Finally, we argue that **Case 3** does not contribute to the approximation ratio as **Case 1** will have an approximation ratio that is at least as large. Recall our initial assumption that ϕ remains as s. This implies that every vertex we have explored to is at least $|\phi t| + \frac{1}{c}\text{dist}(\phi) \geq |st|$ away from t, as otherwise they would have been set to ϕ (described in Section 3.2). As a result, the length of the shortest path from any vertex outside of T_{st} cutting through the congested region must be at least $c|st|$. This is already covered in **Case 1** (see Equation 4.1).

Thus, to minimise the approximation ratio, an important question arises: how can we optimise x to minimise $\max(\text{Equation } 4.1, \text{Equation } 4.2)$? We observe that when $x \to \infty$, Equation 4.1 approaches ∞ while Equation 4.2 approaches 1, encapsulating the trade-off between exploring too conservatively and exploring excessively. This indicates that the minimum should be at the intersection of Equation 4.1 and Equation 4.2. Equating Equation 4.1 and Equation 4.2, we obtain the following expression for x:

$$x = \frac{2c\sin\left(\alpha - \frac{\pi}{6}\right) - c^2 - 1}{2\left(\sin\left(\alpha - \frac{\pi}{6}\right) - c\right)} \tag{4.3}$$

As sides $|sx_0|$ and $|st|$ are fixed in length, we observe that increasing α in Figure 4.16 will only increase $|x_0t|$. Consequently, this will increase the length of the shortest path and lower the approximation ratio. Hence, we substitute in the extreme value 0 for α to get a more simplified expression for x:

$$x = \frac{c^2 + c + 1}{2c + 1} \tag{4.4}$$

As we have balanced Equation 4.1 and Equation 4.2 with x, we can substitute x into either equation to obtain the final approximation ratio. For simplicity, we'll substitute Equation 4.4 into Equation 4.1.

$$\begin{aligned}\frac{2x+2c}{c} = \frac{2(\frac{c^2+c+1}{2c+1})+2c}{c} &= \frac{6c^2+4c+2}{2c^2+c} \\ &= 3 + \frac{c+2}{2c^2+c}\end{aligned} \tag{4.5}$$

Since $\frac{c+2}{2c^2+c}$ is a monotonically decreasing function for $c \geq 1$, let $c = 1$ to maximise Equation 4.5. This results in a 4-approximation of the shortest path when routing positively and concludes our proof for Lemma 7.

We also observe that our algorithm approaches a 3-approximation of the shortest path when we take the limit of $c \to \infty$ in Equation 4.5. This indicates our algorithm will perform better in cases where the congestion factor is higher. □

4.2.2 Negative routing

Without loss of generality, let us assume $t \in \bar{C}_0^s$. We continue proving the second claim of Theorem 1:

LEMMA 8. *The half-plane routing algorithm produces a* $((2\sqrt{3}+5)/\sqrt{3})$*-approximation of the length of the shortest path when routing negatively.*

PROOF. When routing negatively from s, the path in the canonical triangle T_{ts} is upper-bounded by $\frac{5}{\sqrt{3}}c|st|$ as a result of Lemma 1. Let $|st|$ be the unit of length. The lower-bound on the shortest path is $c|st|$. This leads to the following approximation ratio for **Case 1**:

$$\frac{2x + \frac{5}{\sqrt{3}}c}{c} \tag{4.6}$$

In **Case 2**, we consider traversing x down towards t in C_2^s and running out of budget. Let us assume there exists a direct edge to t if we had traversed an arbitrarily small distance of ϵ further. We can express the shortest path length as $x + |x_0t|$.

Analogous to the analysis for routing positively (see Section 4.2.1), we minimise $|x_0t|$ by placing it on the horizontal line through t (see Figure 4.17). Once again, we consider $\angle stx_0$, which can be expressed as $\frac{\pi}{3} + \alpha$.

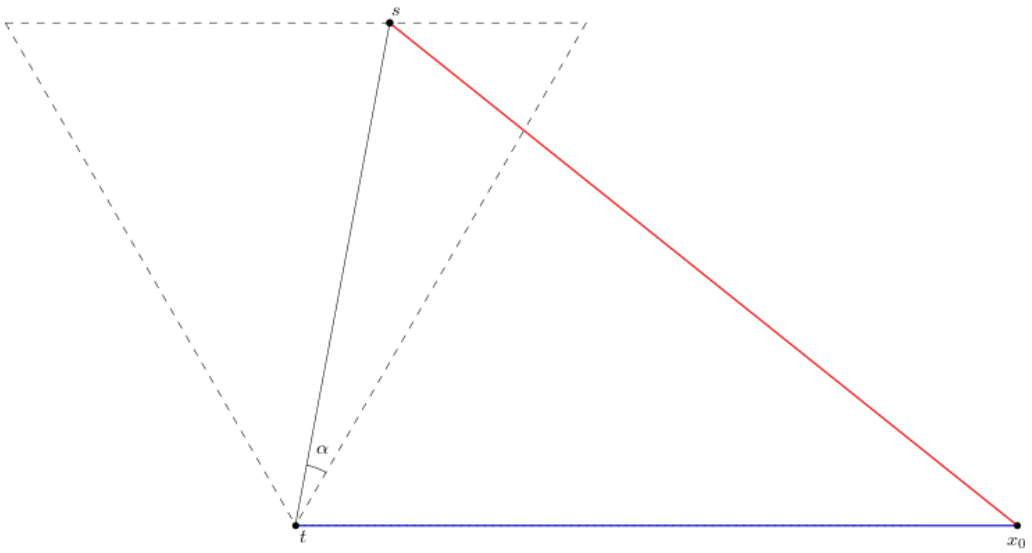


FIGURE 4.17. **Case 2** in negative routing.

Applying Lemma 6 on $\triangle stx_0$, we obtain the following expression for x:

$$x = \sqrt{|x_0t|^2 + 1 - 2|x_0t|\cos(\frac{\pi}{3} + \alpha)}$$

This can be rearranged for a closed-form solution of $|x_0t|$:

$$|x_0t| = \frac{1}{2}\left(\sqrt{2}\sqrt{2x^2 - \sin\left(2\alpha + \frac{\pi}{6}\right) - 1} - 2\sin\left(\alpha - \frac{\pi}{6}\right)\right)$$

This gives us our approximation ratio for **Case 2**:

$$\frac{2x + \frac{5}{\sqrt{3}}c}{x + \frac{1}{2}\left(\sqrt{2}\sqrt{2x^2 - \sin\left(2\alpha + \frac{\pi}{6}\right) - 1} - 2\sin\left(\alpha - \frac{\pi}{6}\right)\right)} \tag{4.7}$$

The same argument made in Section 4.2.1 for why **Case 3** does not contribute to the approximation ratio also applies to negative routing. As a result, we can directly equate Equation 4.6 and Equation 4.7 to find the value of x which produces the smallest approximation ratio. This gives us:

$$x = \frac{2c\sin\left(\alpha - \frac{\pi}{6}\right) + c^2 + 1}{2\left(\sin\left(\alpha - \frac{\pi}{6}\right) + c\right)} \tag{4.8}$$

In Figure 4.17, we observe that $|x_0 t|$ will be minimised if we increase α. Thus, we substitute the extreme value of $\alpha = \frac{\pi}{3}$ into Equation 4.8 to obtain the following expression for x:

$$x = \frac{c^2 + c + 1}{2c + 1} \tag{4.9}$$

Note that we arrive at the same exploration budget as when we substitute in $\alpha = 0$ for the positive routing budget (see Equation 4.3). With Equation 4.6 and Equation 4.7 balanced, we substitute Equation 4.9 into Equation 4.6:

$$\begin{aligned} \frac{2(\frac{c^2+c+1}{2c+1}) + \frac{5}{\sqrt{3}}c}{c} &= \frac{2c^2 + 2c + 2}{2c^2 + c} + \frac{5}{\sqrt{3}} \\ &= 1 + \frac{5}{\sqrt{3}} + \frac{c+2}{2c^2 + c} \end{aligned} \tag{4.10}$$

Since $\frac{c+2}{2c^2+c}$ is a monotonically decreasing function for $c \geq 1$, we consider $c = 1$ to maximise Equation 4.10. This results in an approximation ratio of $(2\sqrt{3} + 5)/\sqrt{3}$ for negative routing and concludes our proof for Lemma 8. □

Theorem 1 follows from Lemma 7 and Lemma 8. □

4.3 Path quality

We now turn our attention to the case where we find an uncongested path from s to t. We make the following claim:

THEOREM 2. *If an uncongested path is found after entering the search phase, the path will be a* 2*-approximation of the shortest path when routing positively and a* 4.4*-approximation when routing negatively. This is the lowest approximation ratio that a local routing algorithm can achieve.*

Let s be the vertex we start the search phase from and v be a vertex on the shortest path which our algorithm will eventually converge back into. Without loss of generality, let the congested region be to the left of s. We refer to the length of the shortest path as σ in the following sections.

4.3.1 Positive routing

Without loss of generality, let $t \in C_0^s$. Since we are entering the search phase from s, the only vertex s is connected to in T_{st} must be in the congested region. Hence, both u and v must be outside of T_{st}.

If the closest vertex to t in C_2^t is below s, our algorithm may erroneously search forwards in the region defined by $\bar{C}_1^s \cap \bar{C}_0^t$, only to be blocked by some congested vertex just before reaching t. Any edges our algorithm's path shares with the shortest path will improve the approximation ratio so we assume that our algorithm's path diverges from the shortest path at s. Let v be the closest vertex to t in C_2^t which is below s. Thus, $\sigma = |sv| + |vt|$. To minimise σ, we place v a small distance below the horizontal boundary of C_2^s. We maximise the length of the path produced by our algorithm by placing u as close to t as possible and then having us return to v.

Since our algorithm stays close to the right boundary of T_{st}, we may follow a zigzag configuration of points such as the one depicted in Figure 4.18. This occurs when our algorithm takes a step in the C_0 cone too close to the right boundary of T_{st} and is blocked by a congested vertex. To minimise our algorithm's progress, we assume the next step in the $\bar{C}_1$ cone is almost horizontal. The following step will traverse as close to the boundary of T_{st} using the C_0 cone and the pattern repeats once more. We observe that each step in the pattern forms an equilateral triangle against the line su, implying that we have to travel two units of distance to move one unit of distance along su. This statement holds regardless of the amount of equilateral triangles in our pattern.

We note that the region defined by $\bar{C}_1^s \cap \bar{C}_0^t$ is also an equilateral triangle, with a maximum side length of $|st|$ (see Figure 4.18). As a result, we have $|su| = |st| = |sv| = |vt|$. To get from s to u, it will take us $2|st|$. Hence, we can express the length of our path as $4|st|$ and $\sigma = 2|st|$. This leads to our algorithm producing a 2-approximation of σ.

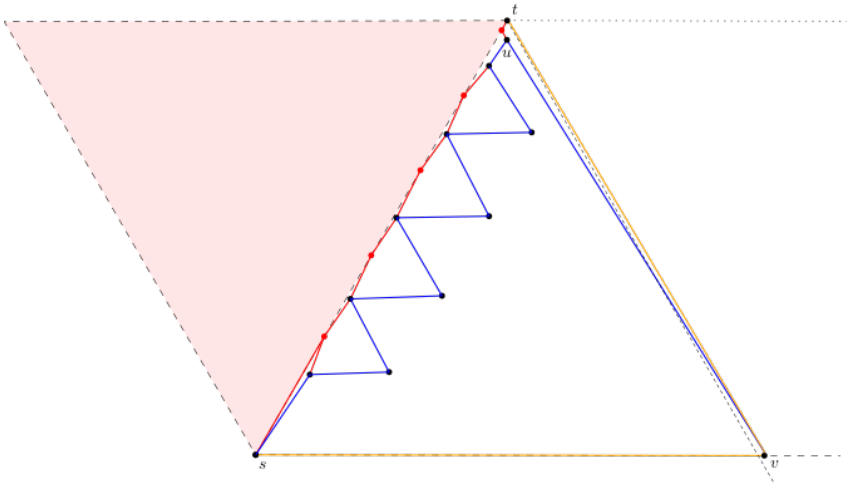


FIGURE 4.18. Orange shows the shortest path and blue shows the suboptimal path our algorithm takes before converging back into the shortest path (positive routing). Edges which are not on either our algorithm's path nor on the shortest path are omitted.

We will now argue that this instance yields the worst-case approximation ratio across all graphs. First, we observe that moving t will decrease the area of $\bar{C}_1^s \cap \bar{C}_0^t$. In turn, this will decrease the length of our algorithm's search path and improve the approximation ratio. We now fix t to consider the positions of u and v. We make use of the following observation in our proof:

OBSERVATION 3. *In an equilateral triangle, the longest distance defined by any two points in the triangle is equal to the side length.*

Let u' be an alternative position for u. If we place u' away from t anywhere within $\bar{C}_1^s \cap \bar{C}_0^t$, we can conclude that $2|su'| + |u'v| \leq 2|su| + |uv|$ due to Observation 3. This results in a shorter path length produced by our algorithm and hence, an improved approximation ratio. If $u' \in C_2^s$, there does not exist any vertices in $\bar{C}_1^s$ and we are immediately searching backwards. Since s is only connected to one vertex in C_2^s, this implies that u' must be on the shortest path. Therefore, placing $u' = u$ will maximise our approximation ratio.

Fixing u to be as close to t as possible, we now consider v', an alternative position for v. Unlike u', the position of v' also affects the length of the shortest path. We first observe that $v' \notin \bar{C}_1^s$. The region of $\bar{C}_1^s$ is divided into two parts: $\bar{C}_1^s \cap \bar{C}_0^t$ and $\bar{C}_1^s \cap C_2^t$. If $v' \in \bar{C}_1^s \cap \bar{C}_0^t$, v' cannot be directly connected to t and will be connected to u instead. On the other hand, if $v' \in \bar{C}_1^s \cap C_2^t$, there is no direct edge from v' to s because u is closer to s than v' is. Hence, v' must be in C_2^s. We observe that v' also cannot be in $C_2^s \cap \bar{C}_0^t$ as it would be connected to u instead of t. This leaves the region defined by $C_2^s \cap C_2^t$ to be the only viable region for v'.

We observe that t, v and v' creates a triangle with $\angle tvv' \geq \frac{2\pi}{3}$. Using Lemma 6, we can express $|v't|$ as $\sqrt{|st|^2 + |vv'|^2 - 2|st||vv'|\cos(\angle tvv')}$. We also observe that s, v and v' form a triangle with $\angle svv' = \frac{5\pi}{3} - \angle tvv'$. Hence, we can express $|sv'|$ as $\sqrt{|st|^2 + |vv'|^2 - 2|st||vv'|\cos(\frac{5\pi}{3} - \angle svv')}$. This is visualised in Figure 4.19.

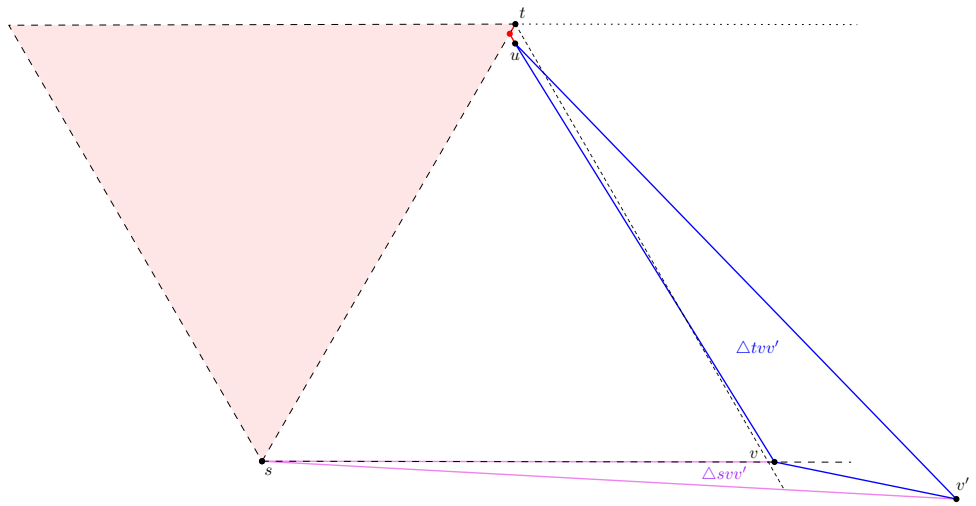


FIGURE 4.19. Visualisations of $\triangle tvv'$ and $\triangle svv'$.

We can express the approximation ratio as:

$$\frac{2|st| + 2\sqrt{|st|^2 + |vv'|^2 - 2|st||vv'|\cos(\angle svv')}}{\sqrt{|st|^2 + |vv'|^2 - 2|st||vv'|\cos(\angle svv')} + \sqrt{|st|^2 + |vv'|^2 - 2|st||vv'|\cos(\frac{5\pi}{3} - \angle svv')}}$$

Formally, we can take the derivative of this function with respect to $|vv'|$ and observe that it is a monotonically decreasing function for $|vv'| > 0$ and $\frac{2\pi}{3} \leq \angle svv' < \pi$. However, computing this can be rather involved and thus, we offer a more intuitive explanation. We first observe that the maximum difference between $\cos(\angle tvv')$ and $\cos(\frac{5\pi}{3} - \angle tvv')$ is $\frac{1}{2}$ at $\angle tvv' = \frac{2\pi}{3}$. This is because the difference between the two cosine functions, $\cos(\angle tvv') - \cos(\frac{5\pi}{3} - \angle tvv')$, is a decreasing function for $\frac{2\pi}{3} \leq \angle tvv' < \pi$. Therefore, we obtain:

$$\sqrt{|st|^2 + |vv'|^2 - 2|st||vv'|\cos(\angle svv')} \leq \sqrt{|st|^2 + |vv'|^2 - 2|st||vv'|\cos(\frac{5\pi}{3} - \angle svv')} + \sqrt{|st||vv'|}$$

Using this bound in the numerator of our approximation ratio, we get:

$$1 + \frac{2|st| + \sqrt{|st||vv'|}}{\sqrt{|st|^2 + |vv'|^2 - 2|st||vv'|\cos(\angle tvv')} + \sqrt{|st|^2 + |vv'|^2 - 2|st||vv'|\cos(\frac{5\pi}{3} - \angle tvv')}} \tag{4.11}$$

Given that $\cos(\angle tvv')$ and $\cos(\frac{5\pi}{3} - \angle tvv')$ are both negative for $\frac{2\pi}{3} \leq \angle tvv' < \pi$, it is clear that the denominator of the second term in Equation 4.11 grows faster than the numerator. As a result, we can infer that this is a decreasing function as we increase $|vv'|$ and vary $\angle tvv'$.

The edge case where $\angle tvv' = \pi$ translates to moving v' downwards while keeping it parallel along the right boundary of $\bar{C}_0^t$. Thus, $\angle svv' = \frac{2\pi}{3}$ and $|uv'| = |uv| + |vv'| = |st| + |vv'|$. We get the following

approximation ratio:

$$\frac{4|st| + 2|vv'|}{\sqrt{|st|^2 + |vv'|^2 + |st||vv'|} + |st| + |vv'|}$$

This function is also a monotonically decreasing function as we increase $|vv'|$. Therefore, this proves that placing v' anywhere other than v will decrease the approximation ratio. The worst-case configuration, therefore, occurs when $v' = v$.

For completeness, we also argue that subdividing the edges and adding more vertices along our path will not affect the maximum approximation ratio. We do this by showing that placing a new vertex w in any position will result in an approximation ratio smaller or equal to 2. First, we show that placing w in any region other than $\bar{C}_1^s \cap \bar{C}_0^t$ will not increase the approximation ratio. There are three regions for w to be in if it is not in $\bar{C}_1^s \cap \bar{C}_0^t$:

(1) If $w \in C_2^s \cap \bar{C}_0^t$ and in T_{sv} is, s would not be connected to v, forcing the shortest path to go through w to get to v. As a result, this would increase the length of the shortest path since the distance from s to v is no longer a straight line distance. Otherwise, w is neither on the shortest path nor on our algorithm's path.

(2) If $w \in C_2^s \cap C_2^t$, w is neither on the shortest path nor on our algorithm's path.

(3) If $w \in \bar{C}_1^s \cap C_2^t$, there exists two subcases where adding w will affect either the length of our path and/or the length of the shortest path:

(3.1) If $w \in C_2^u$ and is closer to u than v is, u would be connected to w. This also results in v being connected to w and can only decrease our approximation ratio since the length of our path remains the same while the length of the shortest path may increase. This is because $|uw| + |wt| \leq |uv| + |vt|$, given $|uw| < |uv|$ and u being close to t.

(3.2) If u is still connected to v, then w could be above u. If w is closer to v than t is in C_0^v, then v is connected to w instead. The distance from v to t is now no longer bounded by a straight-line distance and hence, uniformly increases the distance for both the shortest path and our algorithm's path. We observe that for any fraction $\frac{a}{b}$, where $a > b$ and $x \geq 0$, then $\frac{a}{b} > \frac{a+x}{b+x}$. This shows that adding this additional distance caused by adding w will decrease the approximation ratio.

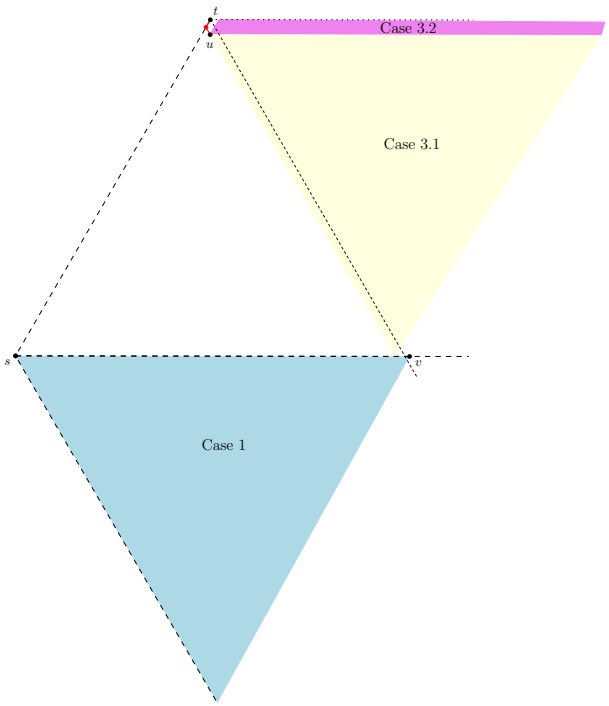


FIGURE 4.20. Cases where adding w will affect the length of the algorithm's path and/or the length of the shortest path.

If w is in $\bar{C}_1^s \cap \bar{C}_0^t$, we argue that the length of the path produced by our algorithm will either remain the same or decrease. Placing w within any of the equilateral triangles in our path from s to u will decrease the length of our path. For example, considering $\triangle v_1v_2v_3$ in Figure 4.21, placing w anywhere within $\triangle v_1v_2v_3$ will cause v_1 to no longer be connected to v_2. This removes v_2 from our algorithm's exploration path. Using Observation 3, we can conclude that $|v_1w| + |wv_2| \leq |v_1v_2| + |v_2v_3|$ and thus, adding w will improve the approximation ratio.

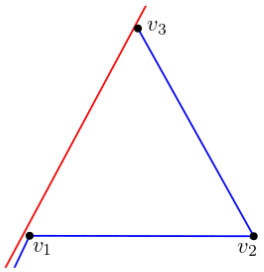
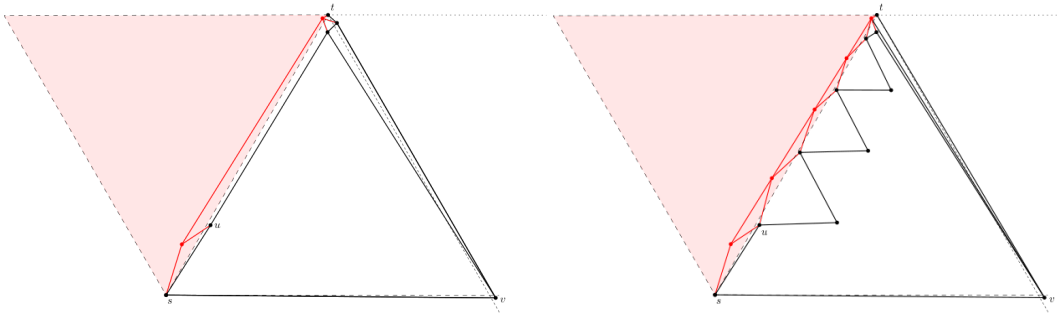


FIGURE 4.21. Configuration of points in the equilateral triangle pattern.

If $w \in C_2^u$, we argue that we can charge any step we take to $|st|$. Since every step we take when searching backwards is in the C_2 cone of our current vertex, which is a positive cone, we can apply Lemma 3. This allows us to conclude that getting from u to v is bounded by $|st|$, regardless of the amount of vertices between them. This completes our proof that the worst-case configuration is as depicted in Figure 4.18.

We will now show that this is the lowest approximation ratio that a local routing algorithm can achieve. Consider Figure 4.22. In the two cases depicted, a local routing algorithm would not be able to distinguish between them. Vertex u is placed in $\bar{C}_1^s \cap \bar{C}_0^t$ arbitrarily close to the right boundary of T_{st} whereas vertex v is placed in $C_2^s \cap C_2^t$ arbitrarily close to the horizontal line going through s.

If a routing algorithm chooses to go to v, the graph could be an instance of Figure 4.22(a), where it would have a path length of $2|st|$ when the path via u has length $|st|$. Conversely, the graph could be an instance of Figure 4.22(b) if we go to u, leading to a path length of $4|st|$ while going through v will give a path length of $2|st|$. This results in the 2-approximation as previously analysed. By adding $\Omega(k)$ points on all edges connected to s, we can obscure any information and ensure that this statement holds for all k-local routing algorithms.



(a) (b)

FIGURE 4.22. Instances where a local routing algorithm would not be able to distinguish between when starting from s. Edges which are not on either our algorithm's path nor on the shortest path are omitted.

4.3.2 Negative routing

Without loss of generality, let $t \in \bar{C}_0^s$ and the congested half-plane be to the left of s. We argue that the worst-case approximation ratio occurs when the shortest uncongested path leaves T_{ts} and re-enters $T_{ts} \cup A$. However, we observe that re-entering and exiting $T_{ts} \cup A$ can be costly to our exploration budget. Hence, we construct a graph configuration where it is difficult to locally determine when to enter $T_{ts} \cup A$.

Let both the shortest uncongested path and our path exit T_{ts} at s and let the shortest path re-enter $T_{ts} \cup A$ at some vertex $w \in A$, which has an arbitrarily short path to t. By alternating congested and uncongested vertices in $T_{ts} \cup A$, we force our algorithm to immediately exit $T_{ts} \cup A$ if we entered at any point before w. This makes it such that our algorithm is not able to continue making any progress towards t while in $T_{ts} \cup A$. In addition, we ensure that the lowest vertex we can see is usually uncongested. This wrongly indicates to our algorithm that there could be an uncongested path in A and causes us to consistently subtract from $\mathcal{R}_2$. In turn, this increases the frequency with which we would have to re-enter A. To maximise the length of our path, we assume when our algorithm runs out of $\mathcal{R}_2$, we must first traverse to a vertex in A followed by a vertex in T_{ts} (see Figure 4.23).

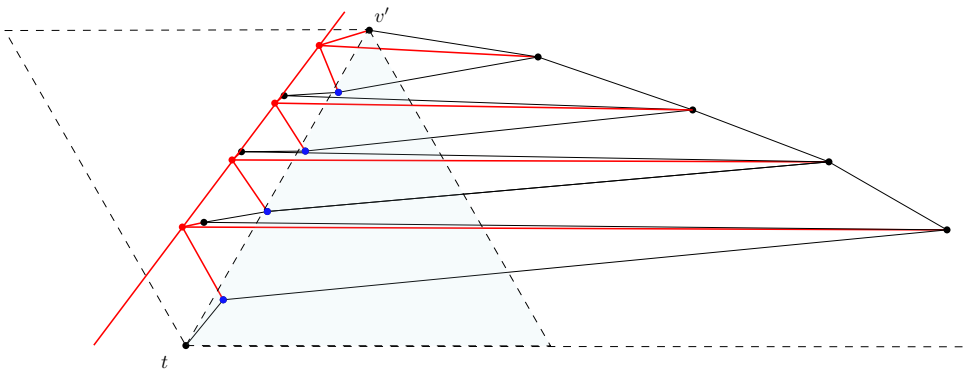


FIGURE 4.23. Worst-case configuration for finding an uncongested path (negative routing). Vertices in A are shown in blue.

We also ensure that consecutive vertices on the exploration path outside of $T_{ts} \cup A$ are arbitrarily close to one another. This prevents our algorithm from progressing towards t when leaving $T_{ts} \cup A$, making it pay the cost of entering and leaving $T_{ts} \cup A$ just to get to the same point as it would have if it had not done so. This is currently not depicted in Figure 4.23 as plotting vertices arbitrarily close to one

another on the horizontal axis can cause edges to cluster together, which becomes visually confusing. We introduce the following lemma to bound the distance travelled to re-enter A:

LEMMA 9. *Let s and t form a canonical triangle T_{ts}, with a vertex v' forming region A next to T_{ts}. If a vertex u is a distance X away from v', then $|uw| \leq X + |v't|$ for any vertex $w \in A$.*

PROOF. Let w be an arbitrary vertex in A and u be some vertex which is X away from v'. We begin by observing that region A is always an equilateral triangle. Applying Observation 3, we can conclude that $|v'w| \leq |v't|$. Considering $\triangle v'wu$, we can use the triangle inequality to get:

$$|uw| \leq |uv'| + |v'w| \leq X + |v't|$$

This bounds the distance from any vertex outside of A to any vertex within A. □

We observe that we can bound the path length from any $w \in A$ to an vertex in T_{ts}, which is in C_1^w, with $|v't|$. This is because the side length of the canonical triangle created by w to any vertex in C_1^w is bounded by $|v't|$. To get to the first vertex in T_{ts} from A, our algorithm will only take vertices in in the C_1 cone. Applying Lemma 3, we can bound the length of our path to a vertex in T_{ts} with $|v't|$. Combining this with Lemma 9, this leads to the following claim:

LEMMA 10. *Let s and t form a canonical triangle T_{ts}, with a vertex v' forming region A next to T_{ts}. If a vertex u is a distance X away from v', then $|uw| \leq X + 2|v't|$ for any vertex $w \in T_{ts} \cup A$.*

We previously assumed that our algorithm is forced to immediately exit $T_{ts} \cup A$ after traversing into T_{ts} to prevent it from making any progress towards t. Let us assume we are now at a vertex $u \in T_{ts}$. This means that there are no uncongested vertices in $\bar{C}_0^u$. We now claim that we cannot ever leave $T_{ts} \cup A$ through some vertex in A. We operated under the assumption that we would enter $T_{ts} \cup A$ at some vertex $v \in A$ first. Since we enter T_{ts} through vertices in C_1^v, v must be in $\bar{C}_1^w$. For the sake of contradiction, let us assume there exists a vertex $w \in C_2^u$ which blocks our exit. This means that $v \in C_0^w$ or $v \in \bar{C}_2^w$.

The only way that v and w are not connected is if there exists either some vertex in $T_{ts} \cup A$ that is closer to w in C_0^w or $\bar{C}_2^w$. We observe that the first case cannot occur as v being connected to u means that their canonical triangle is empty. This means that there cannot exist a vertex in T_{ts} that is closer to w than v is. This is shown in Figure 4.24, where the orange region is guaranteed to be empty. The latter case will still result in some alternative vertex $w' \in A$ being connected to v since no vertex in T_{ts} will block it from doing so. As described in Section 3.2.2, our algorithm will proceed to uncongested vertices in A

before entering T_{ts}. Hence, the presence of w allows our algorithm to make progress towards t while in $T_{ts} \cup A$, which contradicts our setup.

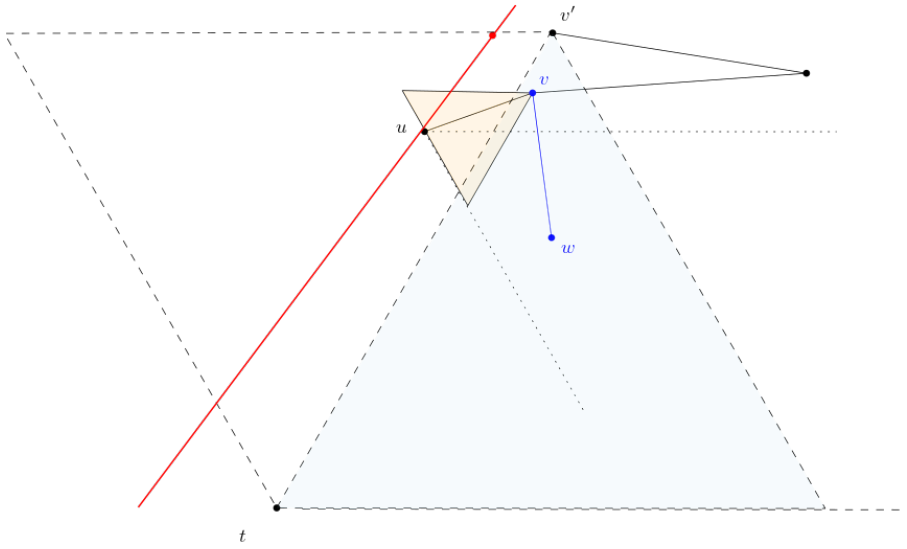


FIGURE 4.24. Configuration where w must be connected to v, given that $w \in C_2^u$ and $v \in \bar{C}_1^u$.

Thus, we can conclude the following:

LEMMA 11. *If our half-plane routing algorithm enters the region defined by $T_{ts} \cup A$ at some vertex in A and reaches some vertex $w \in T_{ts}$, there cannot be any vertices in A which are in C_2^w.*

We get the following corollary of Lemma 11:

COROLLARY 1. *When our algorithm leaves $T_{ts} \cup A$ at some vertex $u \in T_{ts}$ and moves to a vertex w which is a distance of X away from v', $|uw| \leq X + |v't|$.*

PROOF. Lemma 11 asserts that there must be a direct edge from $w \in T_{ts}$ to some vertex outside of $T_{ts} \cup A$. We observe that any vertex in $C_1^{v'} \cap T_{ts}$ and A is within $|v't|$ of v' as a result of Observation 3. Thus, we can apply triangle inequality on $\triangle v'wu$ to get the bound of $X + |v't|$ to travel from any vertex in $C_1^{v'} \cap T_{ts}$ to a vertex which is a distance of X away from v'. □

Let us assume that the shortest path stops at some vertex u, which is a distance of $\mathcal{R}_2^*$ away from v'. Vertex u is connected to w. In the worst-case, our algorithm checks in A at the vertex v right before u, leaves A at u and continues to explore with budget $\mathcal{R}_2 = 1.5|v'u| = \mathcal{R}_2^*$, overshooting the shortest path.

To help bound the repeated entering and exiting of $T_{ts} \cup A$, we prove that the following pattern holds in our given configuration:

LEMMA 12. *Starting from v', our algorithm travels at most* $3\left(\sum_{i=0}^{\log_{2.5}(\frac{\mathcal{R}_2}{|v't|})}(2.5)^i \cdot |v't|\right) + 3\log_{2.5}(\frac{2.5\mathcal{R}_2}{|v't|}) \cdot |v't|$ *when we run out of budget $\mathcal{R}_2$, entering and exiting $T_{ts} \cup A$.*

PROOF. We show this bound holds by induction. $\mathcal{R}_2$ is initially set to $|v't|$, which is our base-case. To run out of $\mathcal{R}_2$, we travel to some vertex u which is $\mathcal{R}_2 = |v't|$ away from v'. Applying Lemma 10 to bound the distance of entering $T_{ts} \cup A$, we can conclude that the distance travelled to re-enter $T_{ts} \cup A$ is bounded by $\mathcal{R}_2 + 2|v't|$. Since we exit T_{ts} at some vertex w arbitrarily close to u, we use Corollary 1 to bound the distance out with $\mathcal{R}_2 + |v't|$. Thus, the total distance travelled is:

$$\mathcal{R}_2 + \mathcal{R}_2 + 2|v't| + \mathcal{R}_2 + |v't| = 6|v't| = 3 \cdot (2.5)^0 \cdot |v't| + 3|v't|$$

This shows that the bound holds for the base-case.

Our inductive hypothesis is as follows: the bound holds for a vertex u which is $\mathcal{R}_2$ away from s, for $\mathcal{R}_2 \geq 2.5|v't|$.

We now prove that this bound holds for the next time we run out of our newly-allocated budget, $\mathcal{R}_2 = 1.5|v'u|$. Assuming we have left $T_{ts} \cup A$, we are now at some vertex which is arbitrarily close to u. From here, we will travel at most a straight line distance of $1.5|v'u|$ away. This indicates we get to some vertex x which is $2.5\mathcal{R}_2$ away from v'. Hence, we are required to prove that the total distance travelled after entering and exiting $T_{ts} \cup A$ is:

$$3\left(\sum_{i=0}^{\log_{2.5}(\frac{2.5\mathcal{R}_2}{|v't|})}(2.5)^i \cdot |v't|\right) + 3(\log_{2.5}(\frac{2.5\mathcal{R}_2}{|v't|}) + 1)|v't|$$

Using Lemma 9 in conjunction with Corollary 1, we conclude that the distance to enter and exit $T_{ts} \cup A$ at x is:

$$1.5\mathcal{R}_2 + 2.5\mathcal{R}_2 + 2|v't| + 2.5\mathcal{R}_2 + |v't| = 6.5\mathcal{R}_2 + 3|v't|$$

Combining this with our inductive hypothesis, we upper-bound the total distance travelled with:

$$6.5\mathcal{R}_2 + 3|v't| + 3\left(\sum_{i=0}^{\log_{2.5}(\frac{\mathcal{R}_2}{|v't|})} (2.5)^i \cdot |v't|\right) + 3\log_{2.5}(\frac{2.5\mathcal{R}_2}{|v't|}) \cdot |v't|$$

$$= 6.5 \cdot 2.5^{\log_{2.5}(\frac{\mathcal{R}_2}{|v't|})} \cdot |v't| + 3|v't| + 3\left(\sum_{i=0}^{\log_{2.5}(\frac{\mathcal{R}_2}{|v't|})} (2.5)^i \cdot |v't|\right) + 3\log_{2.5}(\frac{2.5\mathcal{R}_2}{|v't|}) \cdot |v't|$$

$$< 7.5 \cdot 2.5^{\log_{2.5}(\frac{\mathcal{R}_2}{|v't|})} \cdot |v't| + 3|v't| + 3\left(\sum_{i=0}^{\log_{2.5}(\frac{\mathcal{R}_2}{|v't|})} (2.5)^i \cdot |v't|\right) + 3\log_{2.5}(\frac{2.5\mathcal{R}_2}{|v't|}) \cdot |v't|$$

$$= 3 \cdot 2.5^{\log_{2.5}(\frac{\mathcal{R}_2}{|v't|})+1} \cdot |v't| + 3\left(\sum_{i=0}^{\log_{2.5}(\frac{\mathcal{R}_2}{|v't|})} (2.5)^i \cdot |v't|\right) + 3(\log_{2.5}(\frac{2.5\mathcal{R}_2}{|v't|}) + 1) \cdot |v't|$$

$$= 3\left(\sum_{i=0}^{\log_{2.5}(\frac{\mathcal{R}_2}{|v't|})+1} (2.5)^i \cdot |v't|\right) + 3(\log_{2.5}(\frac{2.5\mathcal{R}_2}{|v't|}) + 1) \cdot |v't|$$

$$= 3\left(\sum_{i=0}^{\log_{2.5}(\frac{2.5\mathcal{R}_2}{|v't|})} (2.5)^i \cdot |v't|\right) + 3(\log_{2.5}(\frac{2.5\mathcal{R}_2}{|v't|}) + 1) \cdot |v't|$$

This concludes our proof by induction. □

Using Lemma 12, we can bound the length of our path in the following form:

$$3\left(\sum_{i=0}^{\log_{2.5}(\frac{\mathcal{R}_2^*}{|v't|})} (2.5)^i \cdot |v't|\right) + 3\log_{2.5}(\frac{2.5\mathcal{R}_2^*}{|v't|}) \cdot |v't| + 4\mathcal{R}_2^* + |v't| \tag{4.12}$$

The final two terms in Equation 4.12 correspond to traversing out an extra $1.5\mathcal{R}_2^*$ before re-entering A at a vertex which is $2.5\mathcal{R}_2^*$ away. We observe that we can bound $|v't|$ with $\mathcal{R}_2^*$. To bound $3\log_{2.5}(\frac{2.5\mathcal{R}_2^*}{|v't|})$, we can solve for the smallest constant a such that $a \cdot \mathcal{R}_2^* \geq 3\log_{2.5}(\frac{2.5\mathcal{R}_2^*}{|v't|}) \cdot |v't|$. We first substitute $y = \frac{\mathcal{R}_2^*}{|v't|}$:

$$a \cdot y \cdot |v't| \geq 3\log_{2.5}(2.5y) \cdot |v't| \tag{4.13}$$

$$a \geq \frac{3\log_{2.5}(2.5y)}{y} \tag{4.14}$$

Setting $f(y) = \frac{3\log_{2.5}(2.5y)}{y}$, we can compute $f'(y)$ to find the function maximum, which occurs at $y = \frac{e}{2.5}$. Hence, we have a maximum at $f(\frac{e}{2.5}) \approx 3.1$. This means that setting $a = 3.1$ is a sufficient upper-bound. We also make use of the following observation:

OBSERVATION 4.

$$\sum_{i=0}^{\log_x(n)} x^i = \frac{xn-1}{n-1}$$

We can now rewrite Equation 4.12 purely in terms of $\mathcal{R}_2^*$:

$$3\left(\sum_{i=0}^{\log_{2.5}(\frac{\mathcal{R}_2^*}{|v't|})} (2.5)^i \cdot |v't|\right) + 3.1\mathcal{R}_2^* + 5\mathcal{R}_2^* \leq 13.1\mathcal{R}_2^* \tag{4.15}$$

Using Lemma 9, we can bound the length of the shortest path with $2\mathcal{R}^* + |v't| \leq 3\mathcal{R}^*$. We note that this is under the assumption of s being placed arbitrarily close to v'. However, we note that our algorithm's path and the shortest path only diverges at some vertex after v'. As a result, the length of both paths will be uniformly increased by $|sv'|$, which would only improve the approximation ratio. Therefore, we get an approximation ratio of $\frac{13.1}{3} \approx 4.4$ when routing negatively.

To justify that this is the worst-case when we find an uncongested path, we also examine the approximation ratio when A does not exist and show that it results in a lower approximation ratio.

Recall that the approximation ratio of a found uncongested path for positive routing was 2 (see Section 4.3.1), produced by considering the case when staying close to the canonical triangle of s and t is sub-optimal. We can construct a similar equilateral triangle to obtain the same approximation ratio. In other words, we consider the case where our algorithm stays close to the boundary of T_{ts} in $\bar{C}_1^t$ until it reaches arbitrarily close to t (see Figure 4.25). For A to be non-existent for the majority of our exploration path, our algorithm should be able to traverse to vertices outside of T_{ts} in either the C_1 or $\bar{C}_0$ cone at every step. To maximise our path length, we construct a similar chain of equilateral triangles as presented in Section 4.3.1. We assume we traverse to some vertex u that is arbitrarily close to t. However, our path is blocked by some congested vertex at u and thus, our algorithm has to travel to $v \in C_2^t$. We also assume that there exists a direct edge from t to v to minimise σ.

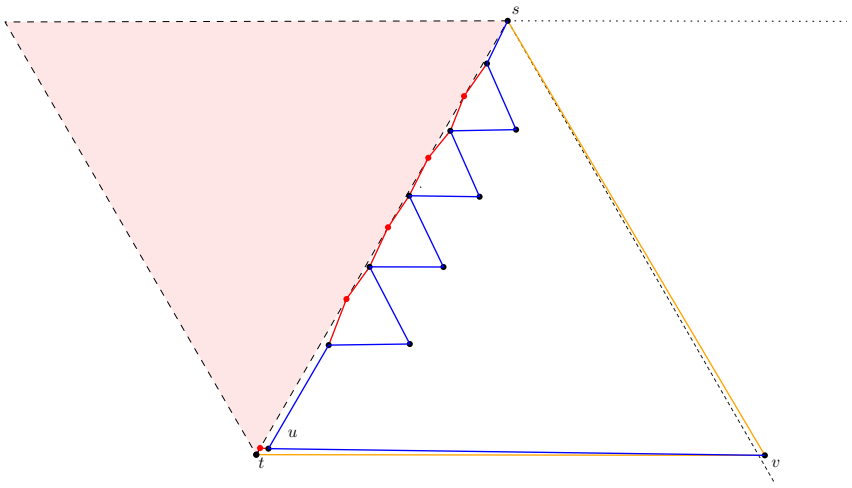


FIGURE 4.25. Orange shows the shortest path and blue shows the suboptimal path our algorithm takes before converging back into the shortest path (negative routing). Edges which are not on either our algorithm's path nor on the shortest path are omitted.

As u becomes arbitrarily close to t, the approximation ratio is expressed as:

$$\frac{2|su| + 2|uv|}{|sv| + |uv|} \tag{4.16}$$

We can apply the same argument made in Section 4.3.1 to show why placing u close to t produces the worst approximation ratio. However, we previously had the constraint that the angle between u and v had to be at least $\frac{\pi}{3}$ because we had to ensure that $v \in C_2^t$. This now no longer applies as we can have $v \in \bar{C}_0^s$. Fixing $\angle suv = \frac{\pi}{3}$, we apply the sine rule to get the following:

$$|sv| = \frac{\sin(\frac{\pi}{3})}{\sin(\frac{2\pi}{3} - \angle usv)} \cdot |su|$$
$$|uv| = \frac{\sin(\angle usv)}{\sin(\frac{2\pi}{3} - \angle usv)} \cdot |su|$$

We can express the approximation ratio as:

$$\frac{2|su| + 2|uv|}{|sv| + |uv|} = \frac{|su|(2 + 2 \cdot \frac{\sin(\angle usv)}{\sin(\frac{2\pi}{3} - \angle usv)})}{|su|(\frac{\sin(\frac{\pi}{3})}{\sin(\frac{2\pi}{3} - \angle usv)} + \frac{\sin(\angle usv)}{\sin(\frac{2\pi}{3} - \angle usv)})}$$
$$= \frac{2\sin(\frac{2\pi}{3} - \angle usv) + 2\sin(\angle usv)}{\sin(\frac{\pi}{3}) + \sin(\angle usv)}$$

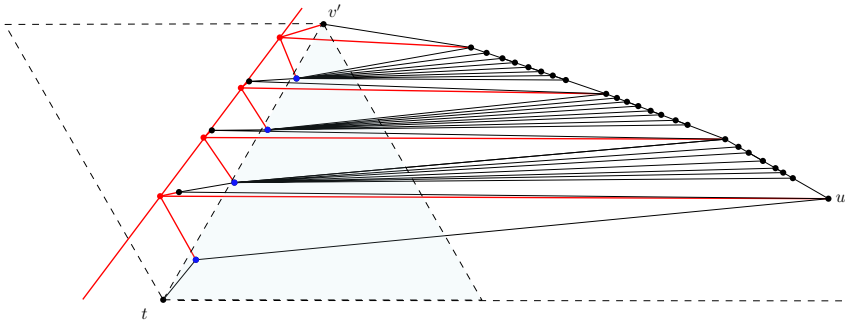
Plotting this function in Desmos, we get a maximum at approximately 2.20 when $\angle usv = 0.43$ radians. This shows that setting $\angle usv = 0.43$ radians will lead to the worst-case approximation ratio under this configuration. As a result, we achieve the following result:

LEMMA 13. *If A does not exist, our algorithm produces a 2.2-approximation of the optimal uncongested path.*

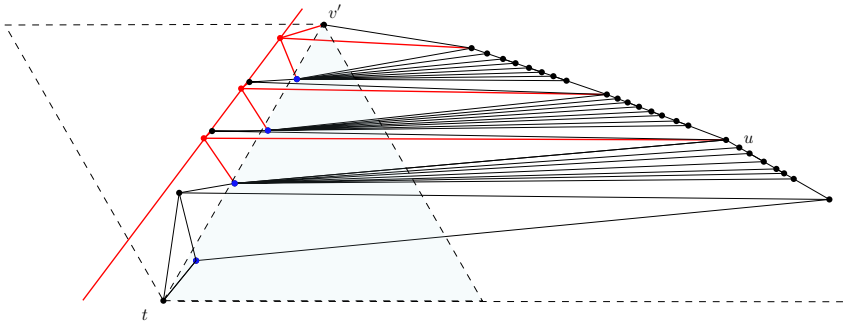
However, this is still lower than the $\frac{13.1}{3}$-approximation found prior.

We proceed by arguing that the $\frac{13.1}{3}$-approximation is a lower-bound for any deterministic k-local routing algorithm. In our current configuration, it is impossible to locally determine how much distance to travel before checking in $T_{ts} \cup A$. With no geometric properties to exploit, any local routing algorithm is required to set some routine interval for checking in $T_{ts} \cup A$ to avoid completely missing an uncongested path. We will refer to this interval as the "check-in interval".

We had previously observed that a linear search can be arbitrarily bad. Consider Figure 4.26, in which the shortest path enters $T_{ts} \cup A$ at vertex u. In both Figure 4.26(a) and Figure 4.26(b), both exploration paths outside of $T_{ts} \cup A$ up to u in Figure 4.26(a) look identical. In Figure 4.26(a), the shortest path enters the $T_{ts} \cup A$ at the very end of the exploration path while in Figure 4.26(b), it enters $T_{ts} \cup A$ at some unknown distance beforehand. We note that the interval length between seeing consecutive congested vertices in $T_{ts} \cup A$ from our exploration path is not necessarily even, giving no useful information for when an algorithm should go in $T_{ts} \cup A$ to verify that there does not exist an uncongested path. As a result, we are always able to construct a graph such that any fixed check-in interval is sub-optimal. For example, if we set the check-in interval to be $|v'u|$ in Figure 4.26(a), we could be in an instance of Figure 4.26(b), which causes us to overshoot the path. Similarly, setting our check-in interval to be too small can result in a frequent exiting and entering of $T_{ts} \cup A$ if we are in an instance of Figure 4.26(a), wasting our exploration budget. By placing $\Omega(k)$ vertices arbitrarily close to the vertices in A, we can obscure any k-local routing algorithm from gaining any information on if entering $T_{ts} \cup A$ at that particular vertex is advantageous or not.



(a)



(b)

FIGURE 4.26. Instances where a local routing algorithm would not be able to distinguish between when starting from v'.

Since the check-in interval is impossible to optimise without knowledge of the shortest path, as shown in Figure 4.26, the only aspect a routing algorithm can optimise is its strategy for determining when to recheck $T_{ts} \cup A$ if the previous check was unsuccessful in finding a path.

To formally argue that our routing algorithm is a lower-bound for any local routing algorithm, we observe that different routing algorithms can use different strategies for determining the distance from v' of their next check-in. This is either done through some uniform increment or an exponential increment of the distance previously travelled. Any algorithm which interpolates their search with a mix of the two strategies can be expressed as some exponential increment as we can assume that all check-ins from the uniform increment were unsuccessful.

We generalise the pattern found in Lemma 12 to both uniform and exponential increments. If a routing algorithm checks in $T_{ts} \cup A$ once every distance of Y, we can lower-bound the length of its path with:

$$3\left(\sum_{i=1}^{\frac{\mathcal{R}^*}{Y}} Y \cdot i\right) + 3(\frac{\mathcal{R}^*}{Y} + 1) \cdot |v't| = \frac{3\mathcal{R}^*(\mathcal{R}^* + Y)}{2Y} + 3(\frac{\mathcal{R}^*}{Y} + 1) = \Omega(\mathcal{R}^{*2})$$

This shows that checking in $T_{ts} \cup A$ at uniform increments results in a path length which is quadratic in terms of the shortest path. Thus, we can conclude that any routing algorithm using this strategy would have a poorer performance.

We note that our algorithm's method is an exponential search, scaling the check-in interval by 2.5 each time. We now generalise the bound in Lemma 12 to other variants of the exponential search. Let this algorithm begin with a check-in interval of δ, which will then increase to travelling out $Y\delta$ from v' for increasing powers of Y. We get the following expression for the bound on its path length:

$$3\left(\sum_{i=0}^{\log_Y(\frac{\mathcal{R}_2^*}{\delta})} Y^i \cdot \delta\right) + 3\log_Y(\frac{Y\mathcal{R}_2^*}{\delta}) \cdot |v't| + (2Y-1)\mathcal{R}_2^* + |v't| \tag{4.17}$$

$$= \frac{3Y\mathcal{R}_2^* - 1}{Y-1} + 3\log_Y(\frac{Y\mathcal{R}_2^*}{\delta}) \cdot |v't| + (2Y-1)\mathcal{R}_2^* + |v't| \tag{4.18}$$

$$\leq \frac{3Y\mathcal{R}_2^*}{Y-1} + 3\log_Y(\frac{Y\mathcal{R}_2^*}{\delta}) \cdot |v't| + 2Y\mathcal{R}_2^* \tag{4.19}$$

We observe that as Y grows, there is a trade-off between the first and last terms in Equation 4.19. The first term $\frac{3Y\mathcal{R}_2^*}{Y-1}$ decreases as Y increases while the final term $2Y\mathcal{R}^*{}_2$ increases. We observe that the logarithmic term $3\log_Y(\frac{Y\mathcal{R}_2^*}{\delta})$ is a decreasing function as Y increases and δ increases. However, as previously established, optimising δ is infeasible due to its dependence on an unknown $\mathcal{R}_2^*$, Furthermore, it is always possible to construct a graph where $\mathcal{R}_2^*$ is adversarial to the δ we have set. This leaves Y as the only variable left to optimise. We compute the intersection of $\frac{3Y\mathcal{R}_2^*}{Y-1}$ and $2Y\mathcal{R}_2^*$ to find the maximum Y we can set without one of the terms dominating and inflating the result. We find that the intersection of $\frac{3Y\mathcal{R}_2^*}{Y-1}$ and $2Y\mathcal{R}_2^*$ occurs at $Y = 2.5$, which is the value used by our algorithm. Therefore, while no routing algorithm can have an optimal choice of δ across all graphs, we show that our algorithm's choice of Y is optimal. This concludes our proof that no local routing algorithm can do better on all graphs.

CHAPTER 5

Routing around convex polygons

Having now designed and analysed an algorithm for routing around a congested half-plane, the next natural progression is to ask: "can we do the same for congested regions contained within a polygon?" To begin approaching this question, we start with looking at convex polygons.

5.1 The $\Omega(c)$-approximation barrier

We open this chapter with an unfortunate result:

THEOREM 3. *There is no local routing algorithm that can do better than a $\Omega(c)$-approximation in the presence of a congested region in the shape of a convex polygon, where c is the congestion factor.*

PROOF. Without loss of generality, let us assume that $t \in \bar{C}_0^s$. We observe that the shortest path is either a completely uncongested path around the polygon or cuts through the polygon at some point. The former case has only two options: the shortest path is either clockwise or anticlockwise around the polygon. On the other hand, the latter case can have $O(n)$ options as we could enter the polygon at any vertex. The challenge of determining where to enter the polygon from is further compounded by the constraints of using only local information. We observe that in order to tell what portion of a particular path is congested, we need information about the vertices on the other side of the polygon. Any edges connected to a congested vertex will be subject to the congestion factor so a path could either be partially congested or fully congested, depending on the existence of any uncongested vertices on the other side. For example, Figure 5.1(a) depicts a partially congested path due to the presence of an uncongested vertex on the other side of the polygon. On the other hand, Figure 5.1(b) depicts a fully congested path due to there being no uncongested vertices on the other side.

FIGURE 5.1. Examples of two paths, one partially congested and one fully congested.

We define the width of the polygon as the length of the polygon that separates s and t. While the width may provide more insight about what portion of the path is congested, polygons with a width that is significantly smaller than $|st|$ yield little to no information. For the purposes of our argument, we model the congested region in the shape of a thin rectangle. Let us assume that there doesn't exist a path around the polygon and the shortest path is through the congested region at u. Let the shortest path be partially uncongested, such that it leaves the congested region at a vertex v which is along the edge of the polygon. Let v be connected to w, a vertex which is outside of the congested region. From w, let there exist a direct edge from w to t. We construct the graph such that every other vertex along our exploration path is connected to some congested vertex which is at most $|uv|$ away from it (see Figure 5.2). We now express the length of the path from s to t via u as:

$$|su| + c(|uv| + |vw|) + |wt| \tag{5.1}$$

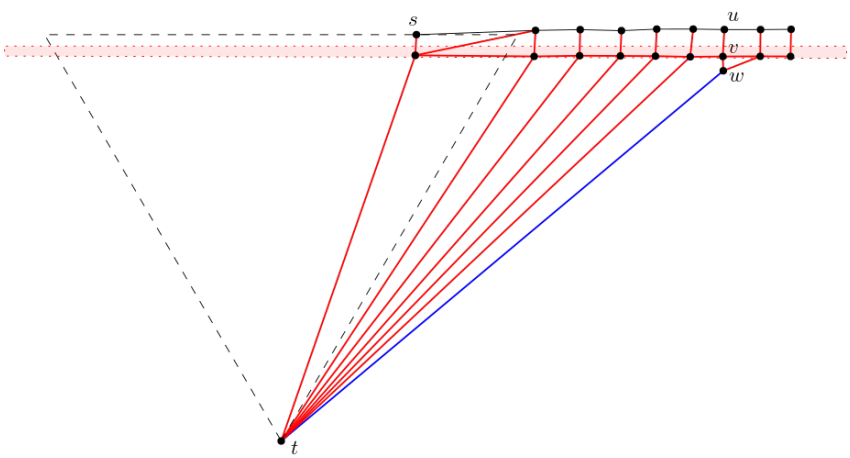


FIGURE 5.2. Graph construction where it is difficult to locally distinguish the shortest path at u.

By placing u arbitrarily close to s, $|su|$ approaches 0. We can minimise the portion of the congested path by also placing u, v and w arbitrarily close to one another, which results in $|wt|$ approaching $|st|$. This allows us to bound Equation 5.1 with $|st| + c(|uv| + |vw|)$. Since we can place u, v and w close to one another, the ratio of $\frac{|uv|+|vw|}{|st|}$ can be reduced to some value less than or equal to $\frac{1}{c}$. As a result, if we do not attempt to find the shortest path and directly route through the congested region in the canonical triangle of s and t, we get the following approximation ratio:

$$\frac{\frac{5}{\sqrt{3}}c|st|}{|st| + c(|uv| + |vw|)} = \frac{\frac{5}{\sqrt{3}}c}{1 + c(\frac{|uv|+|vw|}{|st|})} = \Omega(c)$$

This indicates that if we take the congested path in T_{ts}, the $\Omega(c)$ bound on the approximation ratio holds. Therefore, any algorithm with a better approximation ratio must attempt to find the shortest path. However, when using only local information, all vertices along the search path appear equally viable to be the shortest path. Given that we cannot determine locally which vertex to use to enter the congested region, the only way we can gain information is to go into the congested region until we either find the shortest path or conclude that no partially uncongested path exists. This bounds the length of our exploration before we find the shortest path by $2cn|uv|$. Assuming $\frac{|uv|+|vw|}{|st|} \leq \frac{1}{c}$, we can rearrange to get $|uv| + |vw| \leq \frac{|st|}{c}$. Substituting that into our approximation ratio, we obtain:

$$\frac{2cn|uv|}{|st| + c(|uv| + |vw|)} \geq \frac{2cn|uv|}{2|st|} = n \cdot \frac{c|uv|}{|st|}$$

The approximation ratio of $\Omega(c)$ holds for $n \geq \frac{|st|}{|uv|}$, suggesting that even the effort of checking for the shortest path inherently yields an $\Omega(c)$-approximation. Since both checking and not checking for a partially uncongested path will result in an $\Omega(c)$-approximation, we conclude that no local routing algorithm can achieve a better approximation ratio than this lower bound. □

5.2 Avoiding the congested region

A corollary of Theorem 3 is that ignoring the congestion while routing can yield a reasonably effective solution. Specifically, Bose et al.'s algorithm can be applied directly to any half-Θ_6-graph with congestion as it also results in a $O(c)$-approximation. Regardless, the problem of verifying and finding a path which completely avoids the congested region still stands. The reader may raise the question: can an uncongested path be located if the exploration path is limited by a certain budget?

To answer this, we introduce two variants of a deterministic routing algorithm for routing around congested regions defined by a convex polygon. This algorithm takes a parameter E as input, which limits the distance our algorithm can travel to find an uncongested path. Since routing in uncongested regions has already been covered by Bose et al., we consider this budget E as a limit on the distance *outside* of following Bose et al.'s algorithm.

We begin by making a crucial observation: routing without any knowledge of the shape of the polygon can result in arbitrarily bad exploration paths. When the shape of the polygon is unknown, our algorithm must stay close to the sides of the polygon when attempting to route around. A configuration of points such as the one depicted in Figure 5.3 results in a very poor exploration path as the algorithm must repeatedly approach the red side of the congested region to determine if this particular side has ended.

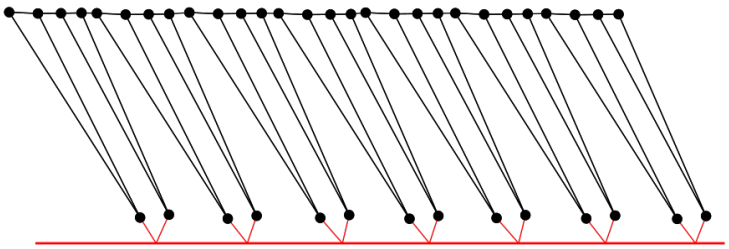


FIGURE 5.3. Example of a bad configuration of points along one side of the polygon.

As a result, we assume we have knowledge of the convex hull of the polygon. We note that our algorithm is not required to store the convex hull in memory as it only uses it at the beginning of the search phase to advise whether to begin searching clockwise or anticlockwise and to compute and store a constant number of extreme points. Analogous to the half-plane routing algorithm described in Chapter 3, both algorithm variants consist of 3 phases: routing, search and return. We will refer to the two variants as A and B. While variants A and B are similar, A will search once on either side of the polygon whilst B will use a technique called the linear spiral search (Baeza-Yates et al., 1993) over both sides of the polygon. We use variant A under the condition that $E \leq \frac{13.1}{2}\mathcal{P}$, where $\mathcal{P}$ is the perimeter of the polygon. We will first present the details of variant A and then outline the differences between it and variant B in Section 5.5.

The routing phase involves the straightforward application of Bose et al.'s algorithm without any modifications. Once it encounters a congested vertex which it would normally traverse to, the algorithm switches to the search phase (described in Section 5.3). Therefore, a separate section for the routing phase will be omitted for brevity.

5.3 Search phase

Let s be the vertex we start from and u be the congested vertex that the optimal routing algorithm would select. We first identify the topmost, bottommost, leftmost and rightmost vertices of the convex hull, denoting these vertices as v_T, v_B, v_L and v_R respectively. Note that vertices are not necessarily distinct. In addition, polygons with horizontal or vertical edges may have both endpoints of that edge qualifying for, say, the topmost vertex. As a result, a maximum of two copies for each of the four extreme points will be maintained i.e. $v_T = u, v'_T = v$, for (u, v) being the topmost horizontal edge in the polygon.

We define an exploration budget $\mathcal{M} = E/3$. Unlike routing along a half-plane, where it is immediately evident that the uncongested path lies on the opposite side of the half-plane, we lack sufficient information to determine if we should start searching clockwise or anticlockwise around the polygon. As a result, our algorithm will search both sides with a budget of $\mathcal{M}$ each.

First, we project s and t onto the polygon to get the points s' and t' on the convex hull (see Figure 5.4). Then, we calculate the perimeter of the polygon from s' to t' in two directions: clockwise and anticlockwise. We begin searching in the direction which yields the smaller perimeter. For example, in Figure 5.4, we would begin searching anticlockwise from s.

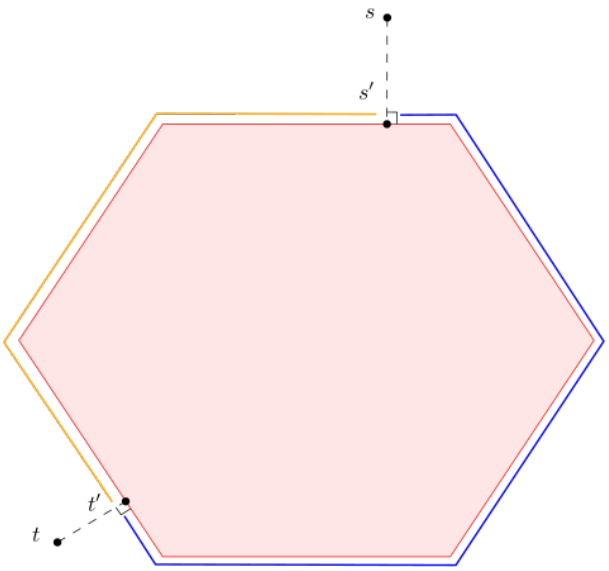


FIGURE 5.4. Projection of s and t onto the convex polygon to obtain s' and t'

We order v_T, v_B, v_L and v_R by occurrence from s' when walking around the polygon in the direction we are currently searching in. Only distinct vertices are kept in this ordering i.e. if $v_B = v_L$, only one of them will be retained. If we encounter t', we end the sequence as it indicates that there are no other

extreme points separating us from t. Without loss of generality, let us assume we are searching in an anticlockwise direction when routing positively and a clockwise direction when routing negatively. We use each of these extreme points as auxiliary targets to get around the polygon, processing each one in the computed ordering. As a result, the routing algorithm may switch between routing positively or negatively as our location with respect to t may change as we route around the polygon. Once we have successfully routed around every extreme point in the given order, we can enter the routing phase to t.

At each extreme point, there are three target cones our algorithm aims to reach. This is determined by the location of the subsequent extreme point in the computed sequence. For the final extreme point, we consider the location of t. Let X denote the cone that contains the next point. The objective is to reach one of the "target cones": X or either of the two cones following X in the opposite direction of the current search i.e. clockwise when searching in an anticlockwise direction. As an example, let us assume we are currently routing anticlockwise around v_R and the next extreme point is v_T which is in $\bar{C}_2^{v_R}$. Thus, the target cones we want to reach when we attempt to route around v_R are $\bar{C}_2^{v_R}$, $C_0^{v_R}$ or $\bar{C}_1^{v_R}$.

This is visualised in Figure 5.5, in which the red region represents the congested polygon. Assuming we are routing anticlockwise around the polygon, the target cones for each extreme point are coloured and labelled. The target cones for v_T and v_B are coloured in blue and are labelled as v_T^* and v_B^* respectively. Similarly, the target cones for v_L and v_R are coloured in orange and labelled as v_L^* and v_R^* respectively. For a vertex in v_B^*, which is $C_0^{v_B} \cup \bar{C}_1^{v_B} \cup C_2^{v_B}$, the first objective is to route to v_R^*, which is $\bar{C}_2^{v_R} \cup C_0^{v_R} \cup \bar{C}_1^{v_R}$. Once in v_R^*, the next goal is to get into v_T^*, and so on. This summarises how our algorithm traverses around the polygon.

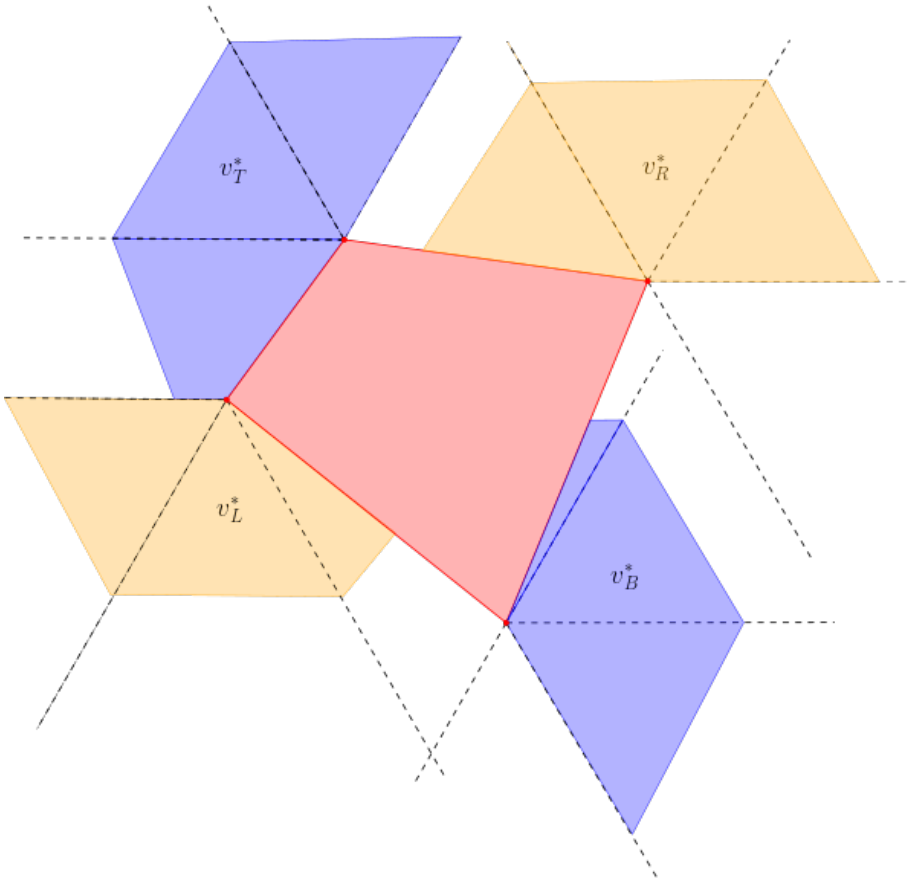
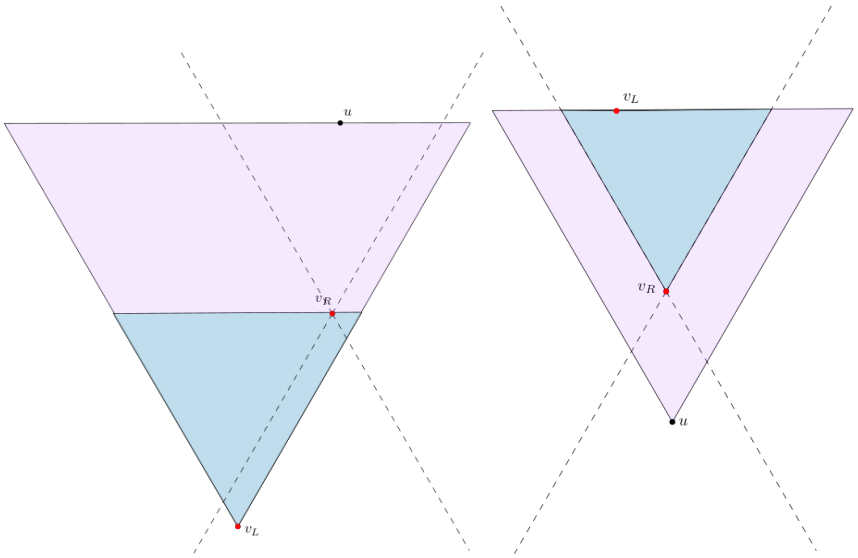


FIGURE 5.5. Target cones around a polygon when moving in an anticlockwise direction.

Using these extreme points to route ensures that at least one of the cones immediately adjacent to the canonical triangle of our current vertex and the extreme point is free from congestion. This approach leverages the convex properties of the polygon and allows us to focus on routing in the cone next to the canonical triangle.

There exists an edge case for if the next extreme point lies in the same cone as the vertex we are currently at. We will continue using the example of routing anticlockwise around v_R, with the next extreme point being v_L. Evidently, X should not be a target cone as our algorithm will simply terminate. Instead, we observe that getting to a vertex u in the cone of v_R directly opposite X is sufficient to ensure that one of the cones immediately adjacent to the cone defining the canonical triangle of u and v_L is not covered by the congested polygon. This is because we observe that the canonical triangle of u and v_L is simply an enlarged version of the canonical triangle of v_R and v_L (see Figure 5.6).



(a) Vertex v_L is in a negative cone of v_R (b) Vertex v_L is in a positive cone of v_R

FIGURE 5.6. Blue represents the canonical triangle of v_R and v_L and purple represents the canonical triangle of a point u and v_L, where u is in the opposite cone of v_L.

As an example, consider Figure 5.7. Assume we are starting at s and we are now routing anticlockwise around the congested polygon, depicted in red, to get to t. We are first required to route around v_R, followed by v_L. However, we observe that v_L lies in the same cone of v_R as s. As a result, getting to a vertex $u \in C_0^{v_R}$ is sufficient in ensuring that the cone next to $T_{v_L u}$, $\bar{C}_2^{v_L}$, is free from congestion.

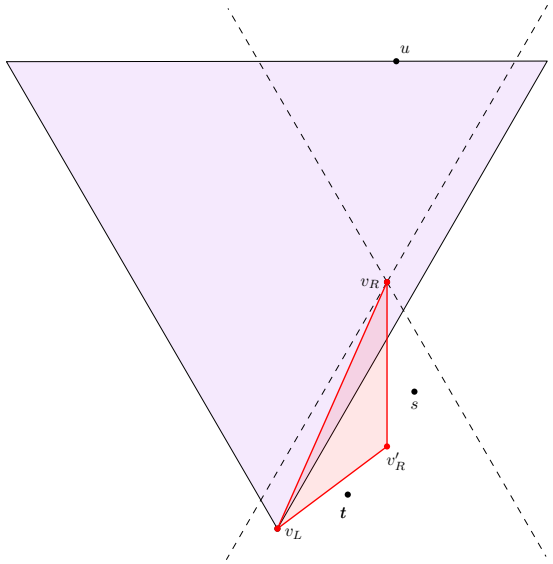


FIGURE 5.7. Example of the edge case, when the next extreme point lies in the same cone as the one we are currently in.

The convexity of the polygon ensures that at least one cone next to the canonical triangle of any two adjacent extreme points will be free from congestion. This is depicted in Figure 5.8, in which we observe that the edges of the congested region are contained within these canonical triangles. Since u creates a canonical triangle with v_L which contains the canonical triangle of v_R and v_L (see Figure 5.6), we can conclude that the cone next to the canonical triangle of u and v_L must be congestion-free. Thus, for this edge case, we redefine the target cones to be the three cones adjacent to X in the opposite direction of our search.

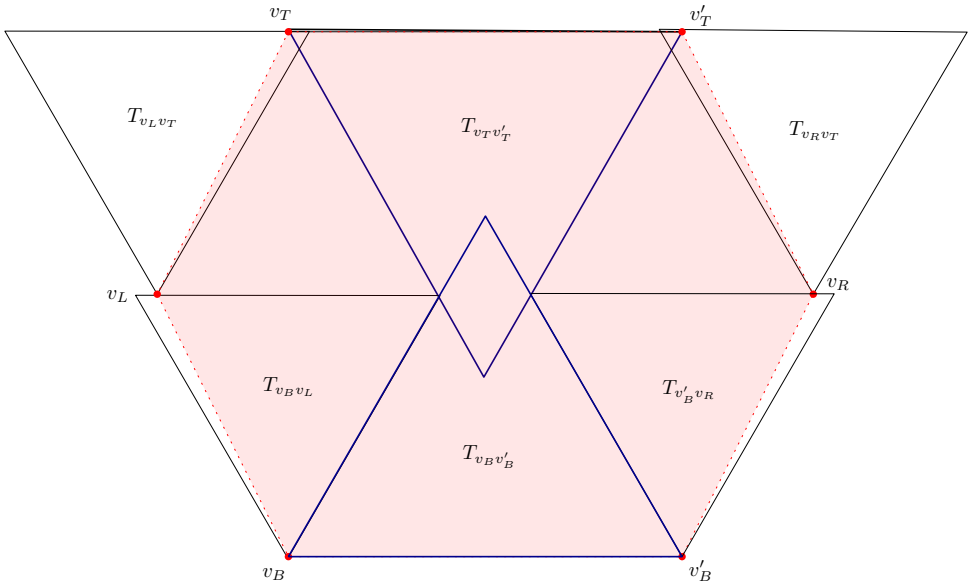


FIGURE 5.8. Polygon with canonical triangles defined by adjacent extreme points.

5.3.1 Notation

We maintain the following variables for each extreme point, with all variables initialised as $\emptyset$ (unless otherwise mentioned):

- γ: the vertex which we start attempting to route around the extreme point from.
- δ: the last vertex we traverse to before leaving the canonical triangle.
- δ': the vertex we use to re-enter the canonical triangle after leaving it.
- f^+: the last vertex in a positive cone of the extreme point before we move into a different cone.
- f^-: the last vertex in a negative cone of the extreme point before we move into a different cone.

- $\hat{f}^+$, $\tilde{f}^+$: the first and last vertices respectively that we traverse to in a positive non-target cone. These variables will only be used when we route negatively.
- v': the vertex we use to construct region A, which was previously introduced and defined in Chapter 3. This variable will only be used when we route negatively and is initialised as $\emptyset$ in that case.
- BACKWARDS: a Boolean to denote if we are in the process of searching backwards. This variable will only be used when we route positively and is initialised as False in that case.
- $\mathcal{B}$: a secondary budget which limits how far we are allowed to traverse outside of the canonical triangle.

Before we traverse to any vertex, we check if the length of the edge to it is within $\mathcal{M}$ and only proceed if that is the case. Once we proceed, we subtract the edge length from $\mathcal{M}$. If there are no vertices within $\mathcal{M}$, we enter the return phase.

We note that we only use the return phase once to get back to s and search the other side of the polygon. To keep track of if we are searching the second side, we maintain an additional variable, SECOND, a Boolean that is initialised as False. In addition, we define $\mathcal{B}$ to be the Euclidean distance from γ to the current extreme point we are processing i.e. $\mathcal{B}_T = |\gamma_T v_T|$

In the following sections, the aforementioned variables will have a subscript to differentiate which extreme point it is associated with i.e. γ_T represents the vertex which we start from when routing around v_T. The majority of these variables are used to facilitate the return phase as it is imperative to follow the same path back to prevent consuming more budget than what was required to get there.

Alongside this, we sometimes describe selecting vertices from a vertex u "in the direction of v", where v is some other vertex. More formally, this refers to vertices in the cone that contains v and its two neighbouring cones.

5.3.2 Positive routing

Let the current extreme point that we want to route around be v_R and $v_R \in C_0^u$, where u is the vertex we are currently at. Since we are searching anticlockwise around the polygon, we want to reach $C_2^{v_R}$. When routing positively, we maintain an additional variable, BACKWARDS, initialised as False. This variable will indicate if we are searching in the opposite direction to v_R as the closest vertex in $C_2^{v_R}$ could be further away than what we expected. We now define four cases:

(1) Inner triangle search

(2) Outer triangle search

(3) Backwards search

(4) Wrap around

We begin in **Case 1** and save $\gamma_R = u$.

5.3.2.1 Case 1: Inner triangle search

This case is for when we are searching for the next extreme point while in T_{uv_R}. We consider vertices in the direction of v_R. If we are connected to at least one vertex in T_{sv_R}, let the first vertex anticlockwise from the right boundary of $T_{\gamma_R v_R}$ be v. In cases where we are also connected to a vertex w which is outside of $T_{\gamma_R v_R}$, select the vertex which forms the smallest unsigned angle against the right boundary of C_0^u (see Figure 5.9).

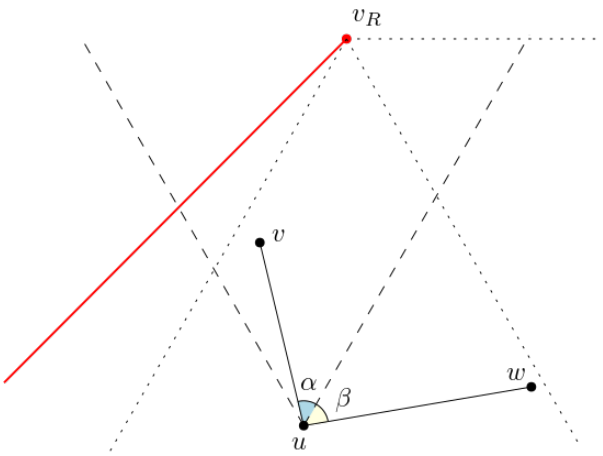


FIGURE 5.9. Comparing α and β to determine whether to traverse to v or w.

The one exception is if we are connected to any vertices in $C_2^{v_R}$ which are also in $T_{\gamma_R v_R}$. Since we want to stay close to v_R, we choose the vertex in $T_{\gamma_R v_R}$ instead.

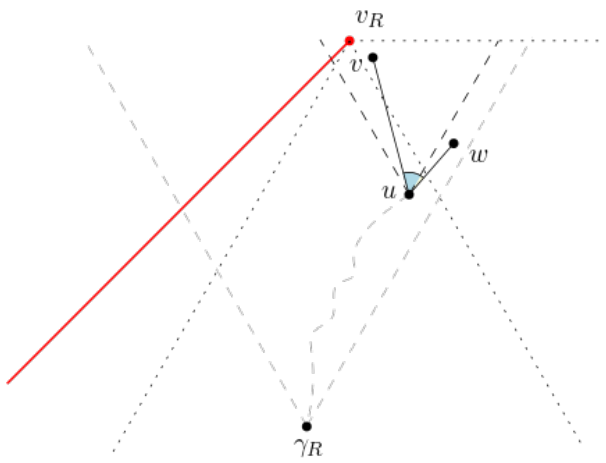


FIGURE 5.10. Example of the exception to selecting vertices with the smallest angle. $T_{\gamma_R v_R}$ and non-important edges are shown in gray.

We remain in this case if we are still in $T_{\gamma_R v_R}$ and repeat the steps above to select the next vertex. Otherwise, should we prepare to traverse to a vertex that is outside $T_{\gamma_R v_R}$, we store our current vertex as δ_R before continuing and entering **Case 2**.

If we reach $C_2^{v_R}$, we move on to processing the next extreme point if $C_2^{v_R}$ is a target cone of v_R and enter **Case 4** otherwise.

In the event that there are no uncongested vertices in the direction of v_R, we enter **Case 3**.

5.3.2.2 Case 2: Outer triangle search

The algorithm enters this case if we are currently outside of $T_{\gamma_R v_R}$ and have not yet arrived at $C_2^{v_R}$ (see Figure 5.11).

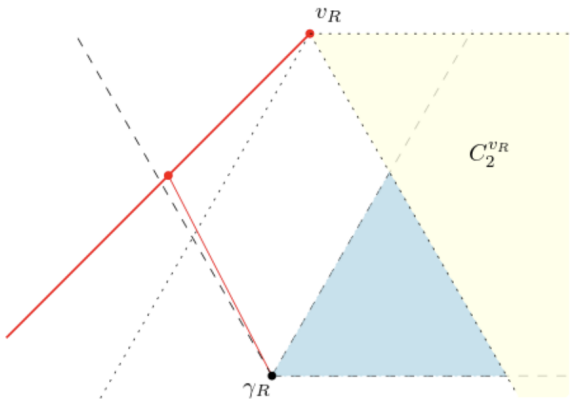


FIGURE 5.11. Blue region shows our current area of exploration. Vertex γ_R is connected to a congested vertex.

Considering vertices in C_0^u or $\bar{C}_1^u$, we select the first vertex which is clockwise from the right boundary of $T_{\gamma_R v_R}$. As we are staying close to the boundary of $T_{\gamma_R v_R}$, we are able to gain information about the path in $T_{\gamma_R v_R}$. If the highest vertex we can see is within the congested region, we do not subtract from $\mathcal{B}$ as we traverse. The basis for this is that seeing a congested vertex implies any path in T_{sv_R} would have to leave T_{sv_R} to avoid the congested region and thus, staying outside of T_{sv_R} is the correct course of action and should not be penalised. Otherwise, if the highest visible vertex is uncongested, it suggests there could be a shorter uncongested path in T_{sv_R} and we should subtract from $\mathcal{B}$ to avoid overshooting it. Consider Figure 5.12 as an example. At vertex u, we are in **Case 2**. Assuming $|uw| > \mathcal{B}$, we will be stopped from proceeding directly to w, which could be arbitrarily far away from u. This prevents us from taking any vertex we see in $C_2^{v_R}$ as it could be unreasonably far away, and we have yet to confirm that an uncongested path does not exist in $T_{\gamma_R v_R}$.

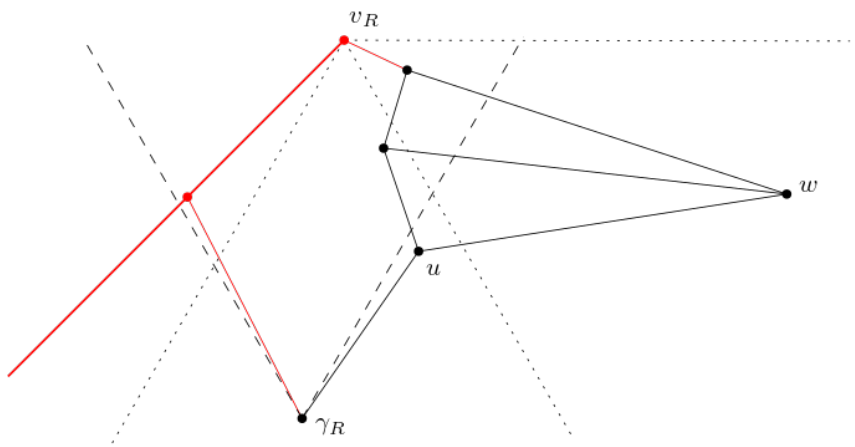


FIGURE 5.12. Example of budget $\mathcal{B}$ preventing our algorithm from traversing to a vertex in $C_2^{v_R}$ that could be arbitrarily far away.

If there are no vertices in $\bar{C}_1^u$ which have an edge length to it within $\mathcal{B}$ but we are connected to an uncongested vertex in T_{sv_R} in C_0^u, proceed to that vertex and set it to be δ'_R. Continue selecting vertices in either C_0^u or $\bar{C}_1^u$ that are closest to the right boundary of $T_{\gamma_R v_R}$ until we reach a vertex in $C_2^{v_R}$. We note that δ'_R could be in $C_2^{v_R}$ and hence, we also perform the steps in the following paragraph.

Before proceeding to any vertex in $C_2^{v_R}$, we save the current vertex as $f_{v_R}^-$. Once in $C_2^{v_R}$, we check if it is one of the three target cones of v_R. If this is the case, we can move on to processing the next point. Otherwise, we enter **Case 4**.

If there are no uncongested vertices to further explore and we have not reached $C_2^{v_R}$, we enter **Case 3**.

5.3.2.3 Case 3: Reverse search

This case signifies that there are no vertices left to explore when moving in the direction of v_R. At this stage, BACKWARDS is set to True and we now repeatedly select vertices in C_2^u. This is done until we are connected to a vertex which is in $C_2^{v_R}$. The current vertex is then marked as $f_{v_R}^-$ and we either move on to the next extreme point (if $C_2^{v_R}$ is a target cone) or enter **Case 4**. If there are no vertices in C_2^u, we enter the return phase (described in Section 5.4). An example of the reverse search is illustrated in Figure 5.13.

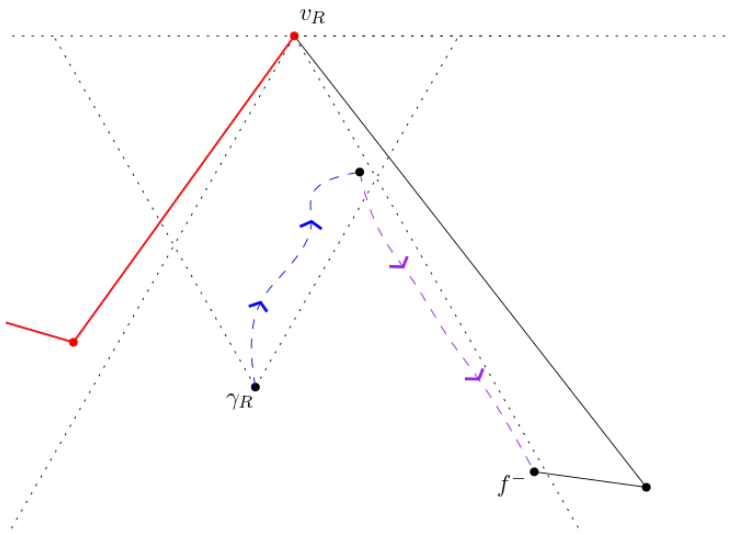


FIGURE 5.13. Example of reverse search. The blue dashed line represents our normal exploration path while the purple dashed line represents our reverse search.

5.3.2.4 Case 4: Wrap around

This case is for when we have reached $C_2^{v_R}$ but it is not a target cone. To traverse, we repeatedly select the first vertex which is anticlockwise from the right boundary of $\bar{C}_0^{v_R}$ in the direction of v_R until we are connected to v_R. Once we are connected to v_R, check if there are vertices in C_0^u or $\bar{C}_2^u$ that are in $C_0^{v_R}$. If there are, set $f^+ = u$.

If we are connected to vertices in both $C_0^{v_R}$ and $\bar{C}_1^{v_R}$, we select the one in $C_0^{v_R}$ as that makes the most progress around the polygon. We can process the subsequent extreme point because $C_0^{v_R}$ is guaranteed to be one of the target cones.

If $\bar{C}_1^{v_R}$ is one of our target cones, we can move onto processing the next point. Otherwise, assuming our current vertex is u, we repeatedly select the first vertex clockwise from the right boundary of $C_0^{v_R}$ in either $\bar{C}_2^u$ or C_0^u. This is done until we see the first vertex in $C_0^{v_R}$. We can now move to processing the next extreme point.

If there are no vertices in either $C_0^{v_R}$ or $\bar{C}_1^{v_R}$ when we are at the closest vertex to v_R in $C_2^{v_R}$, then we conclude that there is no uncongested path around v_R. Thus, we enter the return phase (described in Section 5.4).

5.3.3 Negative routing

Let the current extreme point that we want to route around be v_R and $v_R \in \bar{C}_0^u$, where u is our current vertex. We observe that when routing around a convex polygon, we can encounter the same issues as we did when we were routing around a congested half-plane. In particular, there still remains the issue of arbitrarily bad exploration paths when staying close to the canonical triangle we are currently routing next to (see Figure 3.12). We previously circumvented this problem by constructing a region A and routing outside of $T_{ts} \cup A$. In the context of the convex polygon, this would be $T_{v_R\gamma_R} \cup A$, where A is constructed in the same manner and the secondary budget $\mathcal{B}$ is equivalent in purpose to $\mathcal{R}_2$ in Section 3.2.2. We define three states for when we route negatively:

(1) Inner triangle search
(2) Outer triangle search
(3) Wrap around

We set $\gamma_R = u$ and begin in **Case 1**.

5.3.3.1 Case 1: Inner triangle search

We consider vertices in the direction of v_R. The main distinction between negative and positive routing is that there is a possibility that we can directly enter $C_2^{v_R}$ in $T_{v_R\gamma_R}$ (see Figure 5.14).

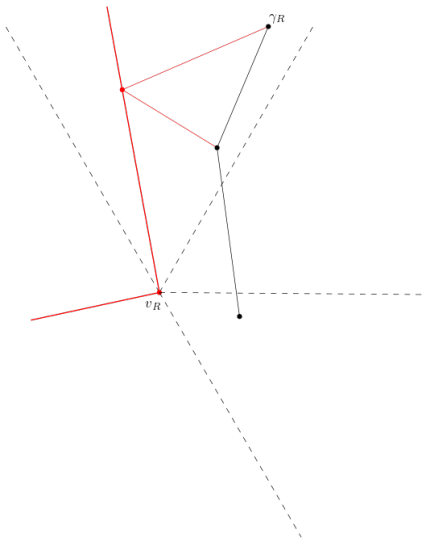


FIGURE 5.14. Being able to directly access $C_2^{v_R}$ in $T_{v_R\gamma_R}$.

We begin searching in the direction of v_R. Instead of selecting the vertex that forms the smallest angle against the right boundary of the canonical triangle, we select uncongested vertices in $T_{v_R\gamma_R}$ which are closest to the right boundary of $T_{v_R\gamma_R}$ and only leave $T_{v_R\gamma_R}$ if there are no uncongested vertices left. If eventually we are connected to v_R in $T_{v_R\gamma_R}$ and can see at least one vertex in $C_2^{v_R}$, we save our current vertex as $f_{v_R}^+$ before selecting the vertex that is closest to v_R in $C_2^{v_R}$. If $C_2^{v_R}$ is a target cone, we move on to processing the next extreme point. Conversely, we enter **Case 3**.

If we have to leave $T_{v_R\gamma_R}$ at some vertex w, we save $\delta_R = w$ and enter **Case 2**.

5.3.3.2 Case 2: Outer triangle search

Considering vertices in the direction of v_R, we select the first vertices closest to the boundary of $T_{v_R\gamma_R} \cup A$. We also subtract from $\mathcal{B}$ following the same criteria outlined in Section 5.3.2.2, with the addition of considering vertices in A as well. If we manage to reach $C_2^{v_R}$, we save $f_{v_R}^- = u$ and move on to processing the next extreme point if $C_2^{v_R}$ is a target cone and **Case 3** otherwise.

In the case where we run out of budget $\mathcal{B}$, we have to re-enter $T_{v_R\gamma_R} \cup A$. However, what could occur is a repeated exiting and entering of $T_{v_R\gamma_R} \cup A$, which wastes the overall exploration budget. In Chapter 3, we depicted this configuration of alternating congested and uncongested vertices with region A (see Figure 4.23). To offer a different perspective, we depict a configuration where A does not contain any vertices, as shown in Figure 5.15. This is to emphasise that we could directly enter T_{v_Rs} without having to go through some vertex in A. As we follow our exploration path, shown in blue, it is clear that going into $T_{v_R\gamma_R}$ at every uncongested point constitutes a highly inefficient use of our exploration budget.

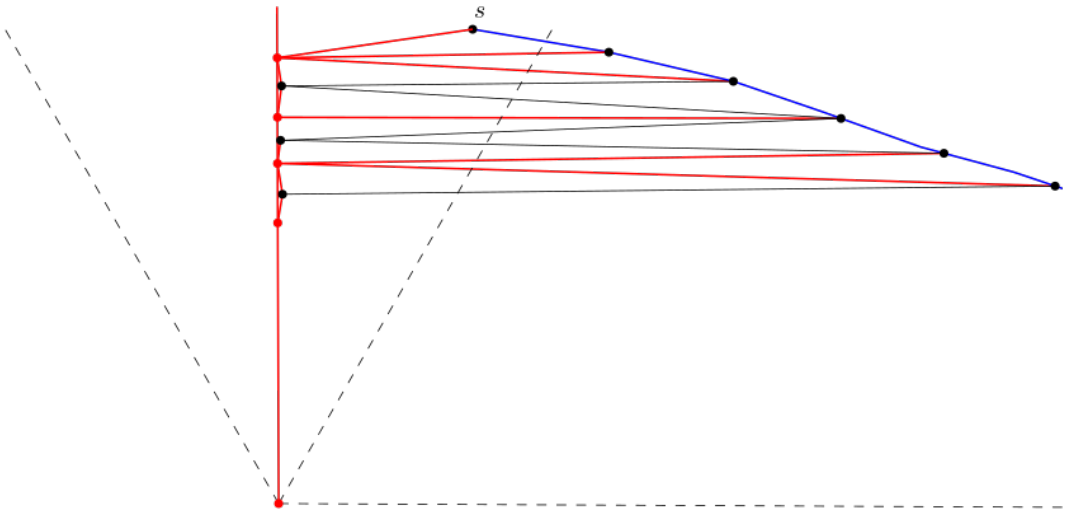


FIGURE 5.15. Extreme case of seeing uncongested vertices in $T_{v_R s}$ which yield no uncongested path.

If we run out of budget $\mathcal{B}$, we re-enter $T_{v_R\gamma_R} \cup A$ at the lowest uncongested vertex w we are connected to in either C_1^u or $\bar{C}_0^u$, setting $\delta'_R = w$. We proceed by following the same steps outlined in Section 3.2.2. To avoid redundancy, we refer readers back to Section 3.2.2 for details. While the approach for selecting vertices on our exploration path remain consistent, we now highlight some differences, primarily related to bookkeeping:

- If we eventually find ourselves connected to at least one vertex in $C_2^{v_R}$, save our current vertex as $f_{v_R}^+$ if we are in $T_{v_R\gamma_R}$ and as $f_{v_R}^-$ if we are in A. Then, we traverse to the vertex that is closest to v_R in $C_2^{v_R}$. Then, we either enter **Case 3** if $C_2^{v_R}$ is not a target cone or otherwise, move on to processing the next extreme point.
- If we are forced to leave $T_{v_R\gamma_R} \cup A$ due to a lack of uncongested vertices to follow, we reset $\delta'_R = \emptyset$ as we wrongly re-entered $T_{v_R\gamma_R} \cup A$ and traversing back into $T_{v_R\gamma_R} \cup A$ is not necessary for the return phase.

Reallocating budget to $\mathcal{B}$ follows the same procedure as for $\mathcal{R}_2$ in Section 3.2.2 as we update $\mathcal{B} = \min(1.5|tv'|, \mathcal{M})$.

In the case where there are no vertices to traverse to outside of $T_{v_R\gamma_R} \cup A$ but there exist uncongested vertices in $T_{v_R\gamma_R} \cup A$ which are below our current vertex, we also enter $T_{v_R\gamma_R} \cup A$ at the lowest uncongested vertex w and set $\delta'_R = w$. We then repeat the steps outlined above.

5.3.3.3 Case 3: Wrap around

We are in this case if we have not yet reached one of the target ones but are in a different positive cone of v_R. We now want to find the first vertex in $\bar{C}_0^{v_R}$ or $C_1^{v_R}$ since both cones are guaranteed to be valid target cones. Using the canonical triangle of v_R and $\hat{f}_{v_R}^+$, we repeatedly select the first vertex anticlockwise from the right boundary of $\bar{C}_0^{v_R}$ in the direction of v_R. At each vertex u we traverse to, we check C_1^u and $\bar{C}_2^u$ for any vertices in $\bar{C}_0^{v_R}$ or $C_1^{v_R}$. If we see a vertex that meets that criteria, we save our current vertex as $\tilde{f}_{v_R}^+$ before traversing to it; we can now move on to processing the next extreme point. If given the choice between traversing to a vertex in $\bar{C}_0^{v_R}$ or $C_1^{v_R}$, we select the vertex in $C_1^{v_R}$ as that makes the most progress around the polygon.

If we are still in $C_2^{v_R}$ and are connected to v_R without finding any valid vertices to traverse to, we can conclude that no uncongested path exists and enter the return phase (described in Section 5.4). This is because all vertices in $C_1^{v_R}$ must be in either C_1^u or $\bar{C}_2^u$ at this current point; being unable to see any vertices indicates that there does not exist any vertices in $C_1^{v_R}$. Figure 5.16 shows this, in which the yellow region is guaranteed to be empty if u is not connected to any vertex in it.

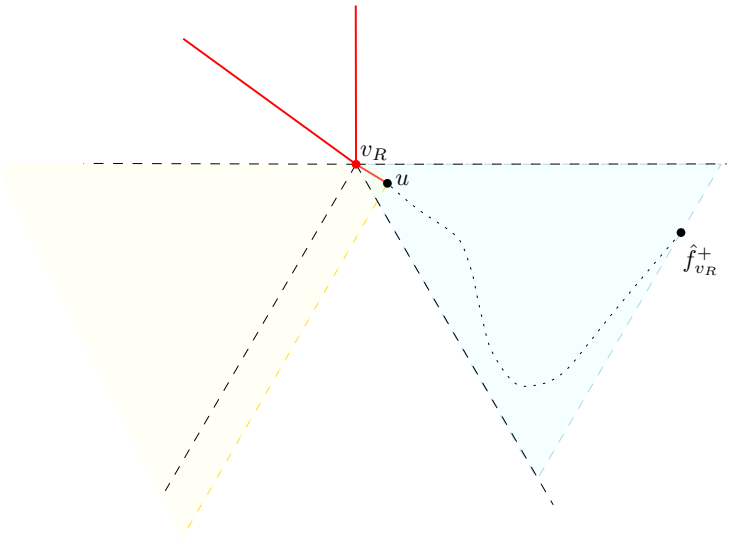


FIGURE 5.16. The canonical triangle of v_R and $\hat{f}_{v_R}^+$ is depicted in blue and our algorithm's exploration path is represented by the dotted black line. The yellow region depicts the region $C_1^u \cup \bar{C}_2^u$ which is below the horizontal line going through v_R.

5.4 Return phase

We enter this phase if we want to return back to s due to exhausting the exploration budget or if there are no viable vertices to further explore. We begin by checking the status of SECOND. If SECOND is True, we may conclude our search since we have already explored the other side of the polygon. We now return False to signify that we were unsuccessful in finding an uncongested path around the polygon within the given budget. Otherwise, if SECOND is False, we need to return to s to continue searching the other side.

To do so, we first consider the extreme points in reverse order of how we were originally considering them i.e. if we were originally searching for v_T, v_R, v_B, we now consider v_B, v_R, v_T. We will refer to this ordering as O. Then, we check what vertex we were in the process of routing towards and what direction around the polygon we were searching in before entering the return phase.

Without loss of generality, let us assume we are currently at a vertex u and we were in the process of routing towards v_R. There are three cases which we could currently be in:

(1) We are in the canonical triangle of γ_R and v_R.
(2) We are outside of the canonical triangle of γ_R and v_R but in a negative cone of v_R.
(3) We are outside of the canonical triangle of γ_R and v_R but in a positive cone of v_R.

Once we reach s, we reset all variables mentioned in Section 5.3 to $\emptyset$ so that they can be reused for routing on the second side. Let u be the vertex that we are currently at.

5.4.1 Positive routing

Without loss of generality, let us assume $\gamma_R \in \bar{C}_0^{v_R}$ and that we were searching around the polygon in an anticlockwise direction prior to entering the return phase.

5.4.1.1 Case 1

In the search phase, we traversed through $T_{\gamma_R v_R}$ by staying close to its right boundary if we are routing in an anticlockwise direction. As a result, we can retrace our steps by consistently following vertices in the canonical triangle that are closest to the right boundary of $T_{\gamma_R v_R}$ in the direction of γ_R. However,

we need to distinguish between the case where we have never left $T_{\gamma_R v_R}$ and the case where we have re-entered $T_{\gamma_R v_R}$. The variable δ is used to make this distinction.

CASE 1.1: $\delta_R = \emptyset$
CASE 1.2: $\delta_R \neq \emptyset$

Case 1.1: This case implies that we have never left the canonical triangle of γ_R and v_R. Thus, we can continue traversing in the direction of γ_R and staying close to the right boundary of $T_{\gamma_R v_R}$ to get back to γ_R. Once we are at γ_R, we stop if $\gamma_R = s$. Otherwise, we consider the next extreme point in O.

Case 1.2: This case implies that we have re-entered $T_{\gamma_R v_R}$ and that a section of our exploration path is outside of the canonical triangle. Thus, we follow vertices in the canonical triangle that are close to the right boundary until we reach δ_R or δ'_R. Once we are at δ_R or δ'_R, select the closest vertex to the right boundary of $T_{\gamma_R v_R}$ which is outside of $T_{\gamma_R v_R}$ and in either $\bar{C}_0^u$ or C_2^u. We now enter **Case 2**.

5.4.1.2 Case 2

If we are in a negative cone of v_R but we are not in $T_{\gamma_R v_R}$, there are two cases:

CASE 2.1: We are in the same cone as γ_R
CASE 2.2: We are not in the same cone as γ_R

Case 2.1: This means we must be in $\bar{C}_0^{v_R}$. We follow vertices in C_1^u or $\bar{C}_0^u$, choosing vertices that stay close to the right boundary of $T_{\gamma_R v_R}$ until we see either γ_R or δ_R. If we see δ_R, we traverse to it and enter **Case 1**.

We note that the variable BACKWARDS will only influence choices made in $\bar{C}_0^{v_R}$, and will thus only affect this specific subcase. BACKWARDS being True indicates that the closest vertex to v_R in $C_2^{v_R}$ is below γ_R. We backtrack by repeatedly selecting the first vertex which is anticlockwise from the right boundary of $\bar{C}_0^{v_R}$ in $\bar{C}_2^u$. This is done until we can only see congested vertices in the direction of v_R as this would indicate that we are at the vertex where we began our reverse search (see Section 5.3.2.3). Once at this point, we can continue following the same steps described in the first paragraph.

When we reach γ_R, we can move on to the next point in O.

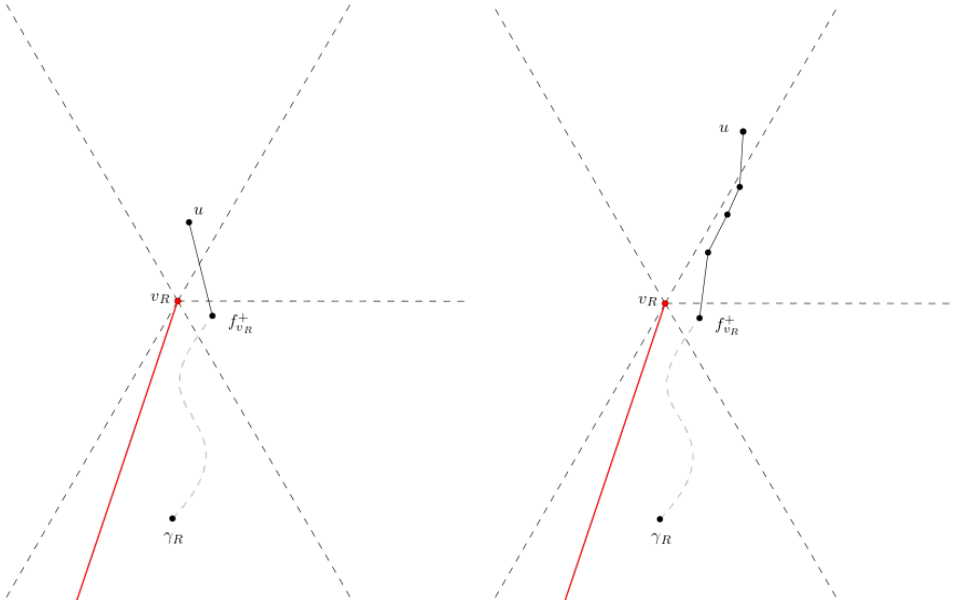
Case 2.2: Being in a different negative cone to γ_R implies that we are in $\bar{C}_1^{v_R}$. Recall that our algorithm may follow the right boundary of $C_0^{v_R}$ upwards if we were attempting to move into $C_0^{v_R}$. This may result

in us being at a vertex which is not immediately visible from $f^+_{v_R}$, the vertex we entered $\bar{C}^{v_R}_1$ from. Out of vertices in $\bar{C}^u_0$ or C^u_2, we backtrack by selecting the first vertex in a clockwise direction from the right boundary of $C^{v_R}_0$ until we are connected to $f^+_{v_R}$. If we see $f^+_{v_R}$, we traverse to it as this will take us back into $C^{v_R}_2$. We can now enter **Case 3**.

5.4.1.3 Case 3

If we are next to $T_{\gamma_R v_R}$, we must be in the region of $C^{v_R}_2$ which is not covered by $T_{\gamma_R v_R}$. Thus, we can backtrack by taking vertices that are closest to the right boundary of $\bar{C}^{v_R}_0$ until we can see $f^-_{v_R}$, the vertex in $\bar{C}^{v_R}_0$ we entered our current cone from. If we see $f^-_{v_R}$, traverse to it and enter **Case 2**.

If we are not next to $T_{\gamma_R v_R}$, we must be in $C^{v_R}_0$. We reached this point either directly from $f^+_{v_R}$ (see Figure 5.17(a)), or through some vertex in $\bar{C}^{v_R}_1$ (see Figure 5.17(b)).



(a) Getting to u from $f^+_{v_R}$ (b) Getting to u from $\bar{C}^{v_R}_1$

FIGURE 5.17. Two methods of getting to $u \in C^{v_R}_0$

In the former case, we can immediately traverse $f^+_{v_R}$ and enter **Case 2**. For the latter, pick the lowest visible vertex in $\bar{C}_1^{v_R}$ and traverse to it. Since we are now in a negative cone of v_R, we enter **Case 2**.

5.4.2 Negative routing

Without loss of generality, let us assume $\gamma_R \in C_0^{v_R}$ and that we were in the process of searching around the polygon in a clockwise direction. Since we also consider region A in our routing, we can rewrite the three cases as the following:

(1) We are in $T_{v_R\gamma_R} \cup A$.
(2) We are outside of $T_{v_R\gamma_R} \cup A$ and in a negative cone of v_R.
(3) We are outside of $T_{v_R\gamma_R} \cup A$ and in a positive cone of v_R.

5.4.2.1 Case 1

Akin to Section 5.4.1.1, we split **Case 1** into two subcases:

CASE 1.1: $\delta_R = \emptyset$
CASE 1.2: $\delta_R \neq \emptyset$

Case 1.1: This case is the most simplistic as it indicates that our algorithm has never left $T_{v_R\gamma_R}$. Hence, we can return to γ_R by travelling to vertices in $T_{v_R\gamma_R}$ along the right boundary of $T_{v_R\gamma_R}$ in the direction of γ_R.

Case 1.2: This case means we must have re-entered $T_{v_R\gamma_R} \cup A$ and hence, $\delta'_R \neq \emptyset$. Regardless of if we are in $T_{v_R\gamma_R}$ or A, we locate δ'_R by selecting vertices that are closest to the right boundary of $T_{v_R\gamma_R}$ in the direction of v'. If we are in T_{v_R}, we prioritise selecting uncongested vertices in T_{v_R} before taking any in A and vice-versa. However, we also ensure that we never pick a vertex which is higher than δ'_R as this may cause us to wrongly traverse past δ'_R. For example, in Figure 5.18, let us assume we are currently at vertex u. While we would normally traverse to vertex v since it is in $T_{v_R\gamma_R}$, vertex v is above δ'_R and is thus, not on our exploration path. Hence, in this instance, we should pick vertex w as it is the only other vertex which is connected to u and is below δ'_R.

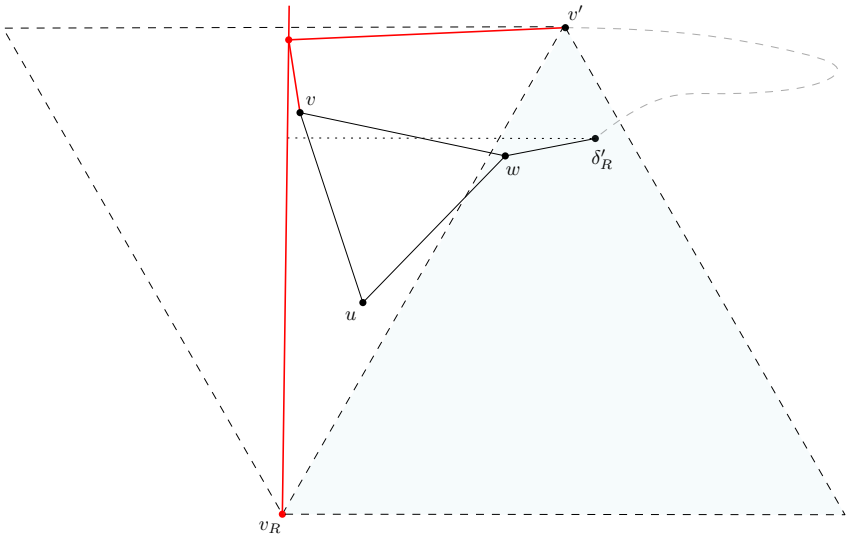


FIGURE 5.18. Locating δ'_R from a vertex $u \in T_{v_R\gamma_R}$. Our exploration path is represented by the curved dashed gray line.

Once we reach either δ'_R, we select the highest vertex in $\bar{C}_1^u$ to traverse to and enter **Case 2**.

5.4.2.2 Case 2

We follow the same criteria and construction instructions for A as outlined in Section 3.2.2. This case corresponds to when we are outside of $T_{v_R\gamma_R} \cup A$ and in a negative cone of v_R. This leads us to one of two possible cases:

CASE 2.1: We are in $\bar{C}_1^{v_R}$.
CASE 2.2: We are in $\bar{C}_0^{v_R}$.

Case 2.1: We can backtrack by selecting vertices that are closest to the boundary of $T_{v_R\gamma_R} \cup A$ in either $\bar{C}_2^u$ or C_0^u until we see γ_R or δ_R in $T_{v_R\gamma_R} \cup A$. If we see γ_R, proceed to it and move on to processing the next point in O. Otherwise, if we see δ_R, proceed to it and enter **Case 1** as that means we are re-entering $T_{v_R\gamma_R} \cup A$.

Case 2.2: We observe that $\bar{C}_0^{v_R}$ is guaranteed to be a target cone of v_R when $\gamma_R \in C_0^{v_R}$. As a result, we should be immediately connected to $\tilde{f}_{v_R}^+$, the vertex in $C_2^{v_R}$ we entered $\bar{C}_0^{v_R}$ from. Thus, we traverse to it and enter **Case 3**.

5.4.2.3 Case 3

If we are in a positive cone which is outside of $T_{v_R\gamma_R} \cup A$, we are in one of two cases:

CASE 3.1: We are in $C_1^{v_R}$.
CASE 3.2: We are in $C_2^{v_R}$.

Case 3.1: In a similar vein to **Case 2.2**, $C_1^{v_R}$ is a guaranteed target cone of v_R and thus, we are immediately connected to $\tilde{f}_{v_R}^+$. Since $\tilde{f}_{v_R}^+$ is located in $C_2^{v_R}$, we enter **Case 3.2** once we traverse to it.

Case 3.2: This case continues to break into two separate subcases as there are two ways we could have reached $C_2^{v_R}$:

CASE 3.2.1: We crossed into $C_2^{v_R}$ from some vertex in $C_0^{v_R}$
CASE 3.2.2: We crossed into $C_2^{v_R}$ from some vertex in $\bar{C}_1^{v_R}$

A similar situation occurs during Section 5.4.1.3 and is depicted in Figure 5.17. We can determine which subcase we are in by checking $f_{v_R}^-$ and $f_{v_R}^+$. Recall that these variables corresponded to the last vertex on our exploration path before leaving a negative cone or positive cone respectively. Thus, if $f_{v_R}^+ \neq \emptyset$, we are in **Case 3.2.1** and if $f_{v_R}^+ = \emptyset$, we are in **Case 3.2.2**.

Case 3.2.1: We should be immediately connected to $f_{v_R}^+$ as to get from $C_0^{v_R}$ directly to $C_2^{v_R}$, we must have traversed to the closest vertex to v_R in $C_2^{v_R}$. As previously established in Section 5.3.3.3, the closest vertex to v_R must be able to see vertices in $C_1^{v_R}$ or $\bar{C}_0^{v_R}$. Therefore, our algorithm would not have needed to move from this vertex to any other vertex in $C_2^{v_R}$. We should traverse to $f_{v_R}^+$ and enter **Case 1**.

Case 3.2.1: If $C_2^{v_R}$ is a valid target cone, we are connected to connected to $f_{v_R}^-$ so we should traverse to it and enter **Case 2**. If not, however, we cannot guarantee that we are connected to $f_{v_R}^-$. Recall that if $C_2^{v_R}$ was not a valid target cone, then we may have traversed to other vertices in $T_{v_R\hat{f}_{v_R}^+}$. In this case, our objective is to find $\hat{f}_{v_R}^+$. This is done by selecting the first vertex anticlockwise from the right boundary of $\bar{C}_0^{v_R}$ in the direction of $\hat{f}_{v_R}^+$. Once we can see $\hat{f}_{v_R}^+$, we can traverse to $f_{v_R}^-$. Recall that $A \in \bar{C}_1^{v_R}$ but is considered as part of **Case 1**. Hence, if $f_{v_R}^- \in A$, enter **Case 1** and **Case 2** otherwise.

5.5 Variant B

We now introduce the second variant of this algorithm, which employs the linear spiral search method introduced by Baeza-Yates et al. (1993). The primary motivation for introducing this variant is to ensure that our algorithm does not product an uncongested path that is arbitrarily worse than the shortest uncongested path around the polygon. Recall that Variant A will explore out a distance of $E/3$ on one side. Without sufficient information, the only statement we can make about the length of the path on the other side of the polygon is that it must be at least $\frac{\mathcal{P}}{2}$. This is because we begin our search on the side of the polygon with the smaller perimeter after projecting s and t on it (see Figure 5.4). Hence, to get around the polygon on the other side, the shortest path must be at least $\frac{\mathcal{P}}{2}$. However, allocating a budget of $E/3$ may not be ideal if E is a fair amount larger than the perimeter of the polygon, specifically when $E \geq \frac{13.1}{2}\mathcal{P}$.

While the main details for the search and routing phase remain consistent, we now conduct a doubling search over both sides of the polygon instead of exploring each side once with $\mathcal{M}$ budget per side. We begin by searching the side with the smaller perimeter with a budget of $\mathcal{P}$. If an uncongested path to t is not found, we return to s before searching $2\mathcal{P}$ on the other side and continue to double this budget on alternating sides until we reach a cap of $E/16$. Our final search on both sides will have a budget of $E/4$ and if a path still has not been found, we return False.

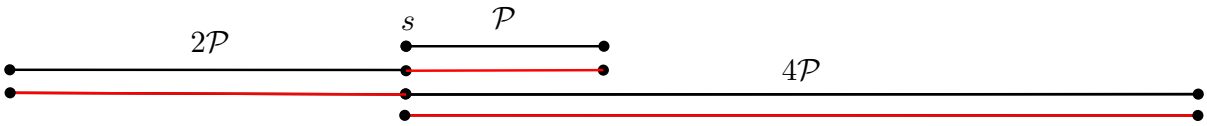


FIGURE 5.19. Visualisation of the linear spiral search strategy.

To optimise our algorithm, we could consider maintaining an additional variable to denote if a particular side has no uncongested vertices to further explore. This naturally implies that continuing to search on that particular side will be fruitless. As a result, this should prompt our algorithm to terminate the doubling search on both sides and reallocate all remaining exploration budget to searching the other side.

5.5.1 Return phase

The main changes occur in the return phase as we will now enter this phase more frequently than before. Previously, in Section 5.4, we used the SECOND variable to limit using the return phase once since we

only explored each side a single time. However, the linear spiral search employed in variant B involves searching both sides of the polygon $O(\log(E))$ times. As a result, each time we enter the return phase, we will check what the current budget $\mathcal{M}$ we have just run out of is. If $\mathcal{M} = E/4$, we now set SECOND to True to indicate that the next search is the final search of distance $E/4$. If we enter the return phase and SECOND is True, we return False to reflect that we were unsuccessful in finding a path.

CHAPTER 6

Convex polygon routing analysis

6.1 Approximation ratio

To evaluate the effectiveness of our routing algorithm presented in Chapter 5, we analyse several cases where we are able to find an uncongested path around the polygon. We make the following claim:

THEOREM 4. *If an uncongested path is found, the convex polygon routing algorithm produces a* 13.1*-approximation of the shortest uncongested path when using Variant A and a* 17.5*-approximation when using Variant B.*

Let s be the vertex we start from. Without loss of generality, let $t \in C_0^s$ when routing positively and $t \in \bar{C}_0^s$ when routing negatively. Routing in purely uncongested sections of the graph are not of interest as they would only improve our approximation ratio. Hence, we assume we immediately encounter the congested convex polygon. Let us assume that only one extreme point v_R separates our current position from t, with t being located arbitrarily close to v_R in either $\bar{C}_0^{v_R}$ (when routing negatively) or $C_0^{v_R}$ (when routing positively). This dually implies that the congested region is to the left of s. We consider the following three cases:

(1) The shortest uncongested path stays completely within the canonical triangle of s and v_R.
(2) The shortest uncongested path is entirely outside of the canonical triangle of s and v_R.
(3) The shortest uncongested path is partially in the canonical triangle of s and v_R.

In the case of negative routing, we will treat region A as part of the canonical triangle. We structure our analysis by first presenting the case that is responsible for the approximation ratios presented in Theorem 4. To prove that this yields the highest approximation ratio, we will then unpack the remaining subcases from the 3 cases above and show that all of them have a smaller approximation ratio.

To maximise the path length produced by our algorithm, we assume that our search is not cut short by a lack of vertices to explore. Furthermore, we assume that we only locate this path to t at the end of our search. For Variant A, this corresponds to finding the path after traversing $E/3$ on the second side, following an earlier search of length $2E/3$. For Variant B, the path would be found on the final search of length $E/4$ after having already explored $3E/4$. We begin our analysis with Variant A.

6.1.1 Variant A

First, we present the case with the highest approximation ratio. When the shortest path is partially in the canonical triangle of s and v_R (**Case 3**) and when we are routing negatively, it is difficult to locally determine when to re-enter the canonical triangle of s and v_R. We observe that an analysis of an identical situation has previously been covered in Section 4.3. Since our algorithm shares the same criteria for selecting vertices on its exploration path as the half-plane routing algorithm when both are routing negatively (see Section 3.2.2), we can apply the findings from Section 4.3 to our current algorithm.

By Theorem 2, we get that routing negatively will be a $\frac{13.1}{3}$-approximation of the shortest uncongested path. In the context of routing around a convex polygon, this is the approximation ratio for the path found on this particular side of the polygon, with respect to the $\frac{E}{3}$ budget assigned to this side only. However, the total length of our exploration path is E as we assumed our algorithm begins by searching the wrong side of the polygon. Thus, the shortest path is actually of length $\frac{E}{3 \cdot \frac{13.1}{3}} = \frac{E}{13.1}$, which results in a 13.1-approximation.

Recall that we use Variant A if $E \leq \frac{13.1}{2}\mathcal{P}$. We now show that this specific threshold was chosen to ensure that Variant A produces a 13.1-approximation. As previously established, the shortest path on the second side of the polygon that we search will be at least $\frac{\mathcal{P}}{2}$. Assuming that we first search the wrong side and then locate this shortest path, we travel the following distance:

$$\frac{2E}{3} + \frac{13.1}{3} \cdot \frac{\mathcal{P}}{2} \leq \frac{13.1}{3} \cdot \mathcal{P} + \frac{13.1}{3} \cdot \frac{\mathcal{P}}{2} = \frac{39.3}{3} \cdot \frac{\mathcal{P}}{2} = 13.1 \cdot \frac{\mathcal{P}}{2}$$

This ensures that our algorithm will remain a 13.1-approximation when using Variant A. We will now examine the remaining cases and show that none of them yield a higher approximation ratio. Within each case, we split the analysis into positive and negative routing as they produce different approximation ratios.

6.1.1.1 Case 1

To start, we make the following observation:

LEMMA 14. *If there exists an uncongested path to t in the canonical triangle of s and v_R, we cannot see any congested vertices from outside of the canonical triangle.*

PROOF. Let us assume that there exists an uncongested $s - t$ path in the canonical triangle of s and v_R, which we will refer to as T_{v_Rs}. Assume we see a congested vertex v from outside of T_{v_Rs} at some vertex u. If we can see v, then (u, v) is an edge that exists. Since u is outside of T_{v_Rs}, this edge must span from the convex hull of the polygon to the opposite side of T_{v_Rs} (see Figure 6.1). The planarity of the half-Θ_6-graph enforces that no edges can cross one another. Therefore, there cannot exist any paths between u and v, which implies that all paths in T_{v_Rs} must go through some part of congested region. This contradicts the assumption of an uncongested path in T_{v_Rs}. Hence, we conclude our proof that we are not able to see any congested vertices if there exists an uncongested path in T_{v_Rs}.

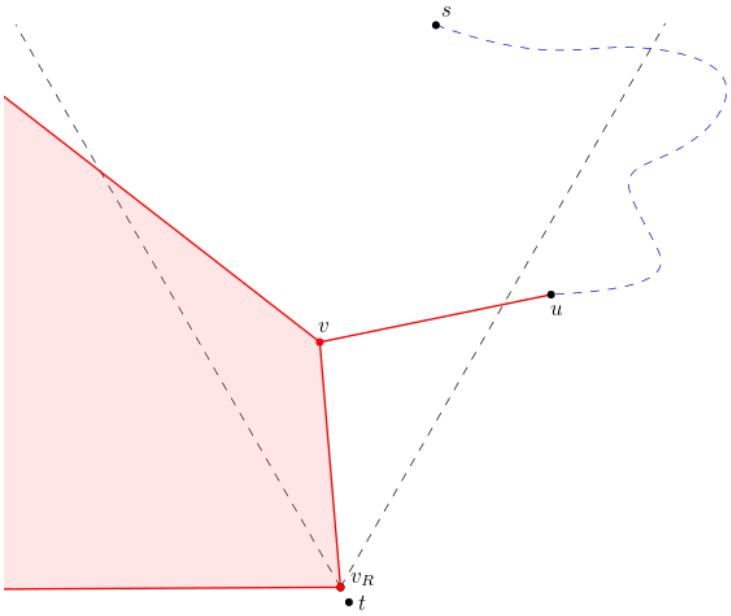


FIGURE 6.1. Example of seeing a congested vertex at a vertex u outside of T_{v_Rs}. The blue dashed line shows our algorithm's exploration path.

□

Positive routing: We consider the case where the shortest uncongested path stays completely within the canonical triangle. Let us assume the shortest path traverses directly from s to some vertex $w \in$

$T_{sv_R} \cap C_2^{v_R}$. Let w also be directly connected to $t \in C_0^{v_R}$ which is arbitrarily close. This bounds the length of the shortest path by $|sv_R|$. Let s also be connected to a vertex u which in $\bar{C}_1^s$. Since staying in T_{sv_R} would mean that we are following the shortest path and improve our approximation ratio, we assume that we leave T_{sv_R} at s. Our algorithm may wrongly choose to go to u as our selection criteria favours selecting the vertex which forms the smallest angle relative to the right boundary of C_0^s. Thus, we let u form a slightly smaller angle than the vertex $w \in T_{sv_R}$. We note that Section 5.3.2.1 describes an exception where we would choose the vertex in T_{sv_R} if it was also in $C_2^{v_R}$, regardless of the angle it forms. To analyse worst-case and ensure our algorithm incorrectly leaves T_{sv_R}, we let some other vertex lie on sw which is just outside of $C_2^{v_R}$.

By Lemma 14, our algorithm can only see uncongested vertices and will subtract from $\mathcal{B} = |sv_R|$ at every step outside of T_{v_Rs}. This constrains the maximum distance that we can traverse out of T_{v_Rs} to be $|sv_R|$. Since $w \in T_{sv_R}$ and u is in the cone immediately adjacent to it, the maximum angle created between sw and su is $\frac{2\pi}{3}$ (see Figure 6.2). We observe that as we increase the angle between sw and su, the length of our algorithm's path will increase. As $\triangle swu$ is an isosceles triangle where $|sw| = |su|$, we derive the following expression for the length of our path when setting $\angle wsu = \frac{2\pi}{3}$:

$$|su| + |uw| = |sv_R| + \frac{\sin(\frac{2\pi}{3})}{\sin(\frac{\pi}{6})} \cdot |sv_R| = (1 + \sqrt{3})|sv_R|$$

Given that the length of the shortest path is $|st| = |sv_R|$, the path that we produced is a $(1 + \sqrt{3})$-approximation. Since we started exploring this side of the polygon with a budget of $E/3$, the length of the shortest path is $\frac{E}{3+3\sqrt{3}}$. This results in a $(3 + 3\sqrt{3}) \approx 8.2$-approximation. We observe that this approximation ratio holds even if the edges were subdivided. This is because the budget of $\mathcal{B}$ enforces a maximum horizontal distance of $|sv_R|$, regardless of the number of vertices on our algorithm's path outside of T_{sv_R}. In addition, placing any vertex in T_{v_Rs} will improve the approximation ratio since it results in a longer shortest path.

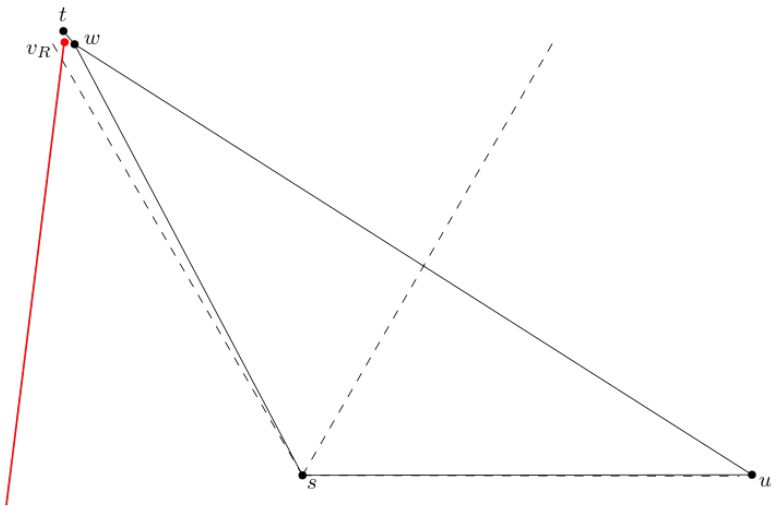


FIGURE 6.2. Worst-case approximation ratio for **Case 1**.

Negative routing: We argue that routing negatively will yield a larger approximation ratio. Recall that when routing negatively, our algorithm will stay within T_{v_Rs} for as long as possible by repeatedly selecting the first vertex anticlockwise from the right boundary of T_{v_Rs}. Since there exists an uncongested path in T_{v_Rs}, our algorithm will remain in T_{v_Rs} for the entirety of its search on this side of the polygon. Applying Lemma 1, we can conclude that the length of our path in T_{v_Rs} is $\frac{5}{\sqrt{3}}|st|$. Since the shortest path is lower-bounded by $|st|$, this leads to a $\frac{5}{\sqrt{3}}$-approximation and when considering the entire budget E, a $\frac{15}{\sqrt{3}} \approx 8.7$-approximation. Given that this a worse approximation ratio than positive routing, we consider the approximation ratio for **Case 1** to be 8.7.

6.1.1.2 Case 2

When the shortest path is entirely outside of the canonical triangle of v_R and s, our algorithm could follow a path either in the canonical triangle or outside of it. Let us first explore the latter case. It was shown earlier in Section 4.3 that staying close to the canonical triangle of s and t when we are outside of it can yield sub-optimal paths. When routing positively, Theorem 2 states that the approximation ratio is 2. We note that if A exists and is non-empty, we obtain the worst-case approximation ratio. For this reason, we prefaced this analysis by stating that we treat A as part of the canonical triangle in the case of negative routing, meaning that the path in **Case 2** is not in A. This has also been analysed in Section 4.3. When routing negatively outside of the canonical triangle of s and t and A does not exist, Lemma 13

states that the approximation ratio is 2.2. This leads to an approximation ratio of 6 and 6.6 for positive and negative routing respectively.

However, we show that this is not the worst approximation ratio for **Case 2**. We now explore the scenario where our algorithm remains in the canonical triangle of v_R and s until there are no uncongested vertices to traverse to.

Positive routing: The lower-bound on the length of the shortest path is $|st| = |sv_R|$ and is attainable by placing a vertex w_1 arbitrarily close to s in $\bar{C}_1^s$ and another vertex w_2 in C_2^s, which the closest vertex to v_R in $C_2^{v_R}$. Vertex w_2 should be connected to $t \in C_0^{v_R}$. Let u be a vertex situated in the leftmost corner of T_{sv_R}, which is the last vertex in T_{sv_R} that our algorithm traverses to before leaving T_{sv_R}. We ensure there is no path from u to w_2 by constructing the graph such that u is blocked by some congested vertex in C_0^u and $\bar{C}_1^u$. This construction is visualised in Figure 6.3.

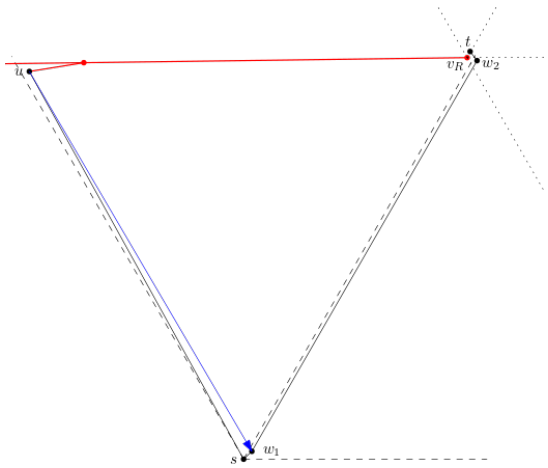


FIGURE 6.3. Worst-case approximation ratio for **Case 2**. The blue arrow indicates the reverse search phase.

This leads to our algorithm entering the reverse search phase and traversing to w_1. Since w_1 is arbitrarily close to s and $u \in T_{sv_R}$, we can apply Lemma 3 and bound the length of $|uw_1|$ with $|sv_R|$. This gives us the following bound on our path length:

$$|su| + |uw_1| + |w_1w_2| + |w_2t| \leq |st| + |st| + |st| = 3|st|$$

This results in an approximation ratio of 3. This means that the length of the shortest path in **Case 2** is $\frac{E}{9}$, which is a 9-approximation. We now briefly argue why this is the worst-case configuration for positive routing, given our algorithm stays inside T_{sv_R}. We observe that moving w_1 anywhere in

$\bar{C}_1^s$ will either decrease $|uw_1|$ or cause the length of the shortest path to grow faster than the length of $|uw_1|$. Let us consider w_1', an alternative position for w_1. Using triangle inequality, we bound $|uw_1|' \leq |su| + |sw_1'| \leq |st| + |sw_1'|$. We use this bound to express the following ratio of the length of our path over the length of the shortest path:

$$\frac{|st| + |st| + |sw_1'| + |w_1't|}{|sw_1'| + |w_1't|} = 1 + \frac{2|st|}{|sw_1'| + |w_1't|} \tag{6.1}$$

Since $|st| \leq |sw_1'| + |w_1't|$, it is evident that Equation 6.1 is at most 3 and occurs when $w_1' = w_1$. As for the position of u, we use Lemma 3 to argue that placing u anywhere other than in one of the corners of T_{sv_R} will result in a shorter path and a smaller approximation ratio. We also observe that no vertex can be in $T_{sv_R} \cap C_0^{w_1}$ as that would cause w_1 to be connected to it instead of w_2.

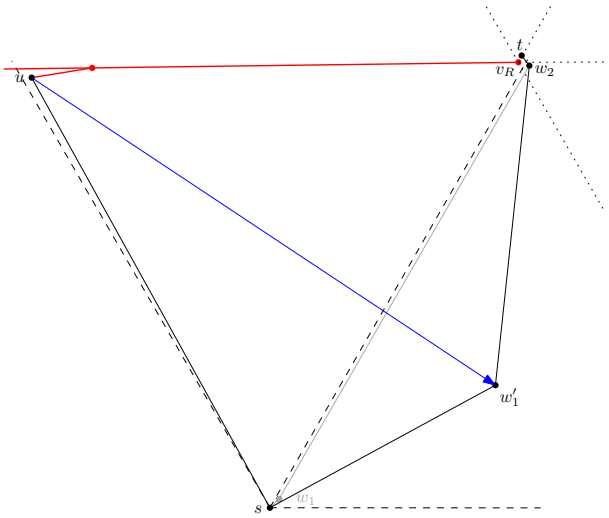


FIGURE 6.4. Alternative positions for w_1 result in a lower approximation ratio.

Negative routing: We show that we are able to attain the same approximation ratio as negative routing in **Case 1**. Once again, we can use Lemma 1 to bound the path within T_{v_Rs} from s to the furthest uncongested vertex we can reach with $\frac{5}{\sqrt{3}}|st|$. Let us assume that our algorithm has reached vertex $u \in T_{v_Rs}$. To attain the bound of $|st|$ on the shortest path outside of T_{v_Rs}, we can place a line of vertices along the right boundary of T_{v_Rs} which directly connect to t. However, for this to occur, we note that s has to be at the right corner of T_{v_Rs}. This indicates that $|st|$ is lower-bounded by the edge length of T_{v_Rs}. As a result, we can use Lemma 2 for a tighter bound of $2.5|st|$ of our path to u.

Notably, we also ensure that there is at least one vertex w_1 on the shortest path which is in $\bar{C}_1^{v_R}$ and is below u. This ensures that our algorithm converges with the shortest path at w_1 instead of the shortest

path converging with our algorithm's path at u (see Figure 6.5). This results in an approximation ratio of 2.5, which is a 7.5-approximation when considering our entire budget of E spent.

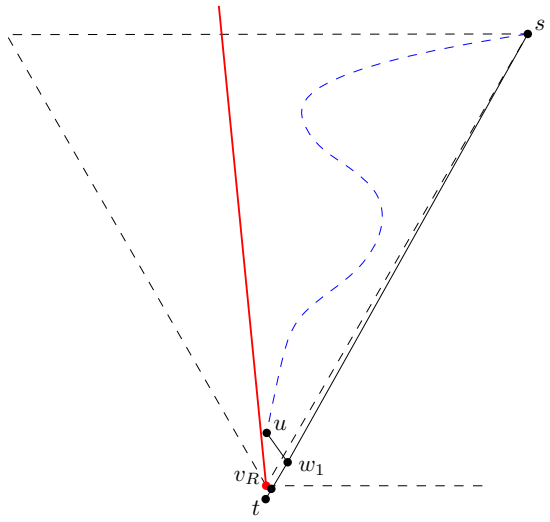


FIGURE 6.5. Worst-case approximation ratio for negative routing in **Case 2**. The blue dashed line represents our algorithm's exploration path to u, the lowest uncongested vertex we can reach.

We now conclude that the approximation ratio for **Case 2** is 9.

6.1.1.3 Case 3

We have established that routing negatively in this case produces our final approximation ratio. For completeness, we will now consider routing positively. We observe that we can use the same analysis as in **Case 2** (see Section 6.1.1.2). This is because if we place w_2 in T_{sv_R}, the shortest uncongested path satisfies leaving and re-entering T_{sv_R}. This results in a 9-approximation, as shown before.

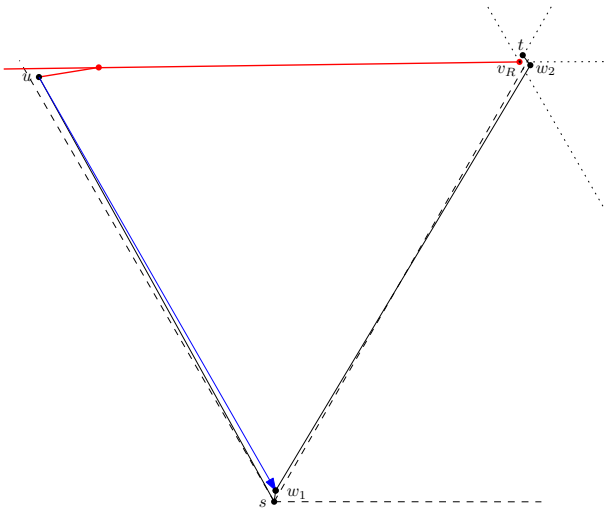


FIGURE 6.6. Worst-case approximation ratio for positive routing in **Case 3**.

We observe that moving v_R, t and w_2 towards the perpendicular bisector of T_{sv_R} will improve the approximation ratio. This will cause u to be closer than w_1 is to w_2 in $C_1^{w_2}$, creating an edge between them and decreasing the length of our algorithm's path.

6.1.2 Variant B

The main difference between Variant A and Variant B is that Variant B does not immediately explore with a budget of $E/3$. Instead, it iteratively increases the budget to a maximum of $E/4$ on both sides. Section 6.1.1 demonstrates that **Case 3** dominates the approximation ratio with a 13.1-approximation. As a result, substituting in $E/4$ instead of $E/3$, we get a shortest path length of $\frac{E}{\frac{52.4}{3}} \approx \frac{E}{17.5}$. As we assume we expend the entirety of E, we get an approximation ratio of $\frac{52.4}{3} \approx 17.5$. This concludes our proof for Theorem 4.

A corollary of Theorem 4 is the following:

THEOREM 5. *If an uncongested path is not found, the shortest uncongested path must have a length of at least* $\frac{3E}{52.4}$.

Instead of finding a path, we assume we run out of our main budget a short ϵ distance away from t. Since Theorem 4 shows that the length of the shortest path must be at least $\frac{3E}{52.4}$. This is the best guarantee our algorithm can make about the length of an undiscovered path.

CHAPTER 7

Conclusion and future work

In this thesis, we studied the problem of local routing in the half-Θ_6-graph under conditions of congestion. We provided a model for congestion in Chapter 2 and presented two deterministic $O(1)$-memory local routing algorithms for it.

We began by assuming the congested region is contained within a half-plane. For this case, we provided a deterministic local routing algorithm in Chapter 3 which is 4-competitive when routing positively and 4.9-competitive when routing negatively. Moreover, we showed that in the case when our algorithm finds an uncongested $s - t$ path, the path we produce is 2-competitive when routing positively and 4.4-competitive when routing negatively. In this case, we proved that this is the best approximation ratio that any k-local routing algorithm can achieve.

However, when the congested region becomes a convex polygon, we proved that no local routing algorithm can do better than an $O(c)$-approximation of the shortest path, where c is the congestion factor. Nonetheless, finding $s - t$ paths that entirely avoid the congested region is an area of ongoing interest, largely motivated by practical use cases. Using our half-plane routing algorithm as a foundation, we designed a deterministic local routing algorithm for this problem and introduce two variants, Variant A and Variant B. This algorithm takes in a parameter which is a constraint on the amount of exploration we can do to find an uncongested path around the polygon. We showed that in the case where an uncongested path is found, Variant A and Variant B produce a 13.1-approximation and 17.5-approximation of the shortest uncongested path respectively. On the contrary, when our algorithm is unable to find a path, we can guarantee that the shortest uncongested path is at least a length of $\frac{E}{17.5}$, where E is the input parameter.

7.1 Future work

There exist other open problems which can be studied in the congested setting. To the best of our knowledge, local routing on other classes of Θ-graphs and spanners in the presence of congestion has not been studied before and thus, could be an avenue for further work. We now offer a few ideas for other directions.

7.1.1 Different models for congestion

In Chapter 2, we modelled congestion by multiplying the edge weight of all edges connected to a congested vertex by some congestion factor c. This means that edges which are not entirely contained within the congested region will still be subject to the multiplicative factor. An alternative idea is to only apply the congestion factor to the portion of the edge within the congested region. This is a looser constraint as it means that an algorithm would be able to traverse to vertices on the border of the congested region from some vertex outside of the congested region without paying any additional cost. We conjecture that this will decrease the approximation ratio for our half-plane algorithm. For the convex polygon algorithm, some minor modifications may be required as a path that traverses to one of the extreme points will now be considered an uncongested path when it previously was not.

Another idea is modelling vertices with different congestion factors instead of all congested vertices sharing the same congestion factor. This problem may be especially challenging if an algorithm only has access to local information as it becomes impossible to bound a path that traverses to non-visible congested vertices.

7.1.2 Routing in the presence of non-convex polygons

The congested region can also be generalised to the shape of a non-convex polygon. Unfortunately, our finding that no local routing algorithm can do better than an $O(c)$-approximation (Theorem 3) still holds. When s and t are both located outside of the convex hull of the polygon, we can simply convert the polygon into a convex polygon using any convex hull algorithm and apply our routing algorithm introduced in Chapter 5. On the other hand, the problem grows increasingly complicated if either s or t is contained in the convex hull of the polygon. We posit that triangulating the polygon and using the algorithm proposed by Korman et al. (2017) for finding the shortest path in polygons may be a good starting point. These vertices in the shortest path may serve as auxiliary vertices which we would want

to route towards. However, we conjecture that using a similar approach to this cannot be achieved with constant memory as there may be $O(n)$ vertices on the shortest path within a polygon.

7.1.3 Defeating the $\Omega(c)$-approximation barrier

Theorem 3 states that no local routing algorithm can do better than an $O(c)$-approximation when the congested region is in the shape of a polygon. While this may be disheartening news for all cases where the shortest path cuts through some section of the polygon, we hypothesize that this limitation can be overcome with some assumptions about the aspect ratio of the polygon or if additional information about the graph is known.

Bibliography

R.A. Baeza-Yates, J.C. Culberson, and G.J.E. Rawlins. 1993. Searching in the plane. *Information and Computation*, 106(2):234–252.

Ron Banner and Ariel Orda. 2005. Multipath routing algorithms for congestion minimization. In Raouf Boutaba, Kevin Almeroth, Ramon Puigjaner, Sherman Shen, and James P. Black, editors, *NETWORKING 2005. Networking Technologies, Services, and Protocols; Performance of Computer and Communication Networks; Mobile and Wireless Communications Systems*, pages 536–548. Springer Berlin Heidelberg.

Luis Barba, Prosenjit Bose, Jean-Lou De Carufel, André van Renssen, and Sander Verdonschot. 2013. On the stretch factor of the theta-4 graph. In Frank Dehne, Roberto Solis-Oba, and Jörg-Rüdiger Sack, editors, *Algorithms and Data Structures*, pages 109–120. Springer Berlin Heidelberg, Berlin, Heidelberg.

Nicolas Bonichon, Prosenjit Bose, Jean-Lou Carufel, Ljubomir Perković, and André Renssen. 2017. Upper and lower bounds for online routing on Delaunay triangulations. *Discrete & Computational Geometry*, 58(2):482–504.

Nicolas Bonichon, Prosenjit Bose, Jean-Lou De Carufel, Vincent Despré, Darryl Hill, and Michiel Smid. 2023. Improved routing on the Delaunay triangulation. *Discrete & Computational Geometry*, 70(3):495–549.

Nicolas Bonichon, Cyril Gavoille, Nicolas Hanusse, and David Ilcinkas. 2010. Connections between theta-graphs, Delaunay triangulations, and orthogonal surfaces. In *International Workshop on Graph-Theoretic Concepts in Computer Science*.

Nicolas Bonichon, Cyril Gavoille, Nicolas Hanusse, and Ljubomir Perković. 2015. Tight stretch factors for L_1- and L_∞-Delaunay triangulations. *Computational Geometry*, 48(3):237–250.

Prosenjit Bose, Jean-Lou De Carufel, Darryl Hill, and Michiel Smid. 2024. On the spanning and routing ratio of the directed theta-four graph. *Discrete & Computational Geometry*, 71:872–892.

Prosenjit Bose, Jean-Lou De Carufel, and Olivier Devillers. 2020. Expected complexity of routing in θ_6 and half-θ_6 graphs. *Journal of Computational Geometry*, 11(1).

Prosenjit Bose, Jean-Lou De Carufel, Stephane Durocher, and Perouz Taslakian. 2017. Competitive online routing on Delaunay triangulations. *International Journal of Computational Geometry & Applications*, 27(04):241–253.

Prosenjit Bose, Jean-Lou De Carufel, Pat Morin, André van Renssen, and Sander Verdonschot. 2016. Towards tight bounds on theta-graphs: More is not always better. *Theoretical Computer Science*, 616:70–93.

Prosenjit Bose, Rolf Fagerberg, André van Renssen, and Sander Verdonschot. 2015a. Optimal local routing on Delaunay triangulations defined by empty equilateral triangles. *SIAM Journal on Computing*, 44(6):1626–1649.

Prosenjit Bose, Darryl Hill, and Aurélien Ooms. 2021. Improved bounds on the spanning ratio of the theta-5-graph. In *Algorithms and Data Structures: 17th International Symposium, WADS 2021, Virtual Event, August 9–11, 2021, Proceedings*, page 215–228. Springer-Verlag, Berlin, Heidelberg.

Prosenjit Bose and Pat Morin. 2004. Online routing in triangulations. *SIAM Journal on Computing*, 33(4):937–951.

Prosenjit Bose, Pat Morin, André van Renssen, and Sander Verdonschot. 2015b. The Θ_5-graph is a spanner. *Computational Geometry*, 48(2):108–119.

Ge-Ming Chiu. 2000. The odd-even turn model for adaptive routing. *IEEE Transactions on Parallel and Distributed Systems*, 11(7):729–738.

K. Clarkson. 1987. Approximation algorithms for shortest path motion planning. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, page 56–65. Association for Computing Machinery, New York, NY, USA.

Edsger W Dijkstra. 1959. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271.

David P. Dobkin, Steven J. Friedman, and Kenneth J. Supowit. 1990. Delaunay graphs are almost as good as complete graphs. *Discrete & Computational Geometry*, 5(1):399–407.

Jingcao Hu and Radu Marculescu. 2004. DyAD: smart routing for networks-on-chip. In *Proceedings of the 41st Annual Design Automation Conference*, DAC '04, page 260–263. Association for Computing Machinery.

Zhen Jiang. 2005. Routing in 2-D meshes: A tutorial. In Yi Pan, Daoxu Chen, Minyi Guo, Jiannong Cao, and Jack Dongarra, editors, *Parallel and Distributed Processing and Applications*, pages 19–20. Springer Berlin Heidelberg, Berlin, Heidelberg.

J. M. Keil. 1988. Approximating the complete Euclidean graph. In *No. 318 on SWAT 88: 1st Scandinavian Workshop on Algorithm Theory*, page 208–213. Springer-Verlag, Berlin, Heidelberg.

Matias Korman, Wolfgang Mulzer, André van Renssen, Marcel Roeloffzen, Paul Seiferth, Yannik Stein, Birgit Vogtenhuber, and Max Willert. 2017. Routing in simple polygons. In *Proceedings of the 33rd European Workshop on Computational Geometry (EuroCG2017)*, pages 17–20. 33rd European Workshop on Computational Geometry : EuroCG 2017, EuroCG 2017 ; Conference date: 05-04-2017 Through 07-04-2017.

Tomás Lozano-Pérez and Michael A. Wesley. 1979. An algorithm for planning collision-free paths among polyhedral obstacles. *Commun. ACM*, 22(10):560–570.

E. F. Moore. 1959. The shortest path through a maze. In *Proceedings of an International Symposium on the Theory of Switching, Part II*, pages 285–292. Harvard University Press, Cambridge, MA.

L. Paul Chew. 1989. There are planar graphs almost as good as the complete graph. *Journal of Computer and System Sciences*, 39(2):205–219.

Ljubomir Perković, Michael Dennis, and Duru Türkoğlu. 2022. The stretch factor of hexagon-Delaunay triangulations. *Journal of Computational Geometry*, 12(2).

Jim Ruppert and Raimund Seidel. 1991. Approximating the d-dimensional complete Euclidean graph. In *Proceedings of the 3rd Canadian Conference on Computational Geometry (CCCG 1991)*, pages 207–210.

Robert Tarjan. 1971. Depth-first search and linear graph algorithms. In *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, pages 114–121.

André van Renssen, Yuan Sha, Yucheng Sun, and Sampson Wong. 2023. The Tight Spanning Ratio of the Rectangle Delaunay Triangulation. In Inge Li Gørtz, Martin Farach-Colton, Simon J. Puglisi, and Grzegorz Herman, editors, *31st Annual European Symposium on Algorithms (ESA 2023)*, volume 274 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 99:1–99:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

Ge Xia. 2013. The stretch factor of the Delaunay triangulation is less than 1.998. *SIAM Journal on Computing*, 42(4):1620–1659.

Guo Xin, Zhang Jun, and Zhang Tao. 2009. A distributed multipath routing algorithm to minimize congestion. In *2009 IEEE/AIAA 28th Digital Avionics Systems Conference*, pages 7.B.2–1–7.B.2–8.

Yihua Xiong and Jerry B. Schneider. 1992. Shortest path within polygon and best path around or through barriers. *Journal of Urban Planning and Development*, 118(2):65–79.