The Polygon Exploration Problem I: A Competitive Strategy

Frank Hoffmann* Christian Icking** Rolf Klein**
Klaus Kriegel*

Abstract

We present an on-line strategy that enables a mobile robot with vision to explore an unknown simple polygon. We prove that the resulting tour is less than 26.5 times as long as the shortest watchman tour that could be computed off-line.

Our analysis is doubly founded on a novel geometric structure called the *angle hull*. This structure is presented in Part II of this paper [13].

Key words: Angle hull, competitive strategy, computational geometry, motion planning, navigation, on-line algorithm, optimum watchman tour, polygon, robot.

1 Introduction

In the last decade, the path planning problem of autonomous mobile systems has received a lot of attention in the communities of robotics, computational geometry, and on-line algorithms; see e. g. Rao et al. [17], Blum et al. [4], and the upcoming surveys by Mitchell [15] in Sack and Urrutia [18] and by Berman [3] in Fiat and Woeginger [10]. We are interested in strategies that are correct, in that the robot will accomplish its mission whenever this is possible, and in performance guarantees that allow us to relate the robot's cost to the cost of an optimal off-line solution or to other complexity measures of the scene.

In this work we are addressing a basic problem in this area. Suppose a mobile robot has to explore an unknown environment modeled by a simple polygon. The

*Freie Universität Berlin, Institut für Informatik, D-14195 Berlin.
**FernUniversität Hagen, Praktische Informatik VI, D-58084 Hagen.

This work was supported by the Deutsche Forschungsgemeinschaft, grant Kl 655/8-3.

A preliminary version [12] of this paper has appeared in the proceedings of WAFR '98.

robot starts from a given point, s, on the polygon's boundary. It is equipped with a vision system that continuously provides the visibility of the robot's current position. When each point of the polygon has at least once been visible, the robot returns to s.[1]

In the on-line polygon exploration problem we ask for a *competitive* exploration strategy that guarantees that the robot's path will never exceed in length a constant *competitive factor* times the length of the optimum watchman tour through s, i. e., of the shortest tour inside the polygon that contains s and has the property that each point of the polygon is visible from some point of the tour. This approach to evaluating the performance of an on-line strategy goes back to Sleator and Tarjan [19]. *A priori* it is not clear whether a competitive exploration strategy exists.

Even the *off-line* version of the polygon exploration problem is not easy. Here we are given a simple polygon and have to compute the optimum watchman tour through a specified boundary point, s. Initially this problem has been suspected to be NP-hard. Chin and Ntafos [7] were the first to provide a polynomial time solution. They have shown how to compute the optimum watchman tour in time $O(n^4)$, where n denotes the number of vertices of the polygon. Later, their result has been improved on to $O(n^2)$ by Tan and Hirata [20].

Carlsson et al. [6] have proven that the optimum watchman tour without a specified point s can be computed in time $O(n^3)$. Furthermore, Carlsson and Jonsson [5] proposed an $O(n^6)$ algorithm for computing the shortest path inside a simple polygon from which each point of the boundary is visible, when start and end points are not specified. In these papers it is always assumed that the range of the robot's visibility is unbounded. Some authors have also studied the case of limited visibility, e. g. Arkin et al. [2] and Ntafos [16].

As to the *on-line* version of the polygon exploration problem, Deng et al. [8] were the first to claim that a competitive strategy does exist. In their seminal paper they discussed a subproblem, incurring a competitive factor of 2016. For the rectilinear case, they gave a complete, and elegant, proof in [9]; here a simple greedy strategy can be applied that performs surprisingly well.

The first proof for the more difficult case of non-rectilinear simple polygons has been given in our conference paper [11]. There we have provided an on-line exploration strategy and sketched a proof that the tour it generates in any polygon is shorter than 133 times the length of the optimum watchman tour. One of the main difficulties with this analysis was in establishing reasonably sharp length estimates for robot paths of complex structure, and in relating them to the optimum watchman tour.

[1] In the absence of holes, the robot has seen each point inside the polygon as soon as it has seen each point on its boundary.

The present paper contains the first complete presentation and analysis of an exploration strategy for simple polygons. As compared to the conference version [11], this full paper has been greatly simplified, and describes a new analysis that is built on an interesting geometric relation between the robot's path and the optimum watchman tour. This relation is expressed in terms of the *angle hull*, a novel geometric structure introduced in Part II of this paper [13]. With these improvements we are able to show that an unknown polygon can be explored, from a given boundary point, s, by a tour at most 26.5 times as long as the shortest watchman tour containing s.

To be a little more precise, our former exploration strategy was based on a certain subdivision of the polygon into subpolygons called rooms. The robot had to explore these rooms one by one; but in doing so it would often enter neighbouring rooms. This made for a complicated and lossful analysis. The strategy presented in this paper avoids this difficulty. Rather than rooms, it uses groups of vertices that are naturally related to the robot's behaviour. As a consequence, local exploration paths can now be charged more easily to the local parts explored.

Our analysis greatly benefits from a new geometric structure we propose to call the *angle hull*. Let D be a simple polygon contained in another simple polygon, P. Then the angle hull, $\mathcal{AH}(D)$, of D consists of all points in P that can see two points of D at an angle of 90°; see Figure 1. The boundary of $\mathcal{AH}(D)$ can be described as the path of a diligent photographer who uses a 90° angle lens and wants to take a picture of D that shows as large a portion of D as possible but no walls of P. Before taking the picture, the photographer walks around D, in order to inspect all possible viewpoints.

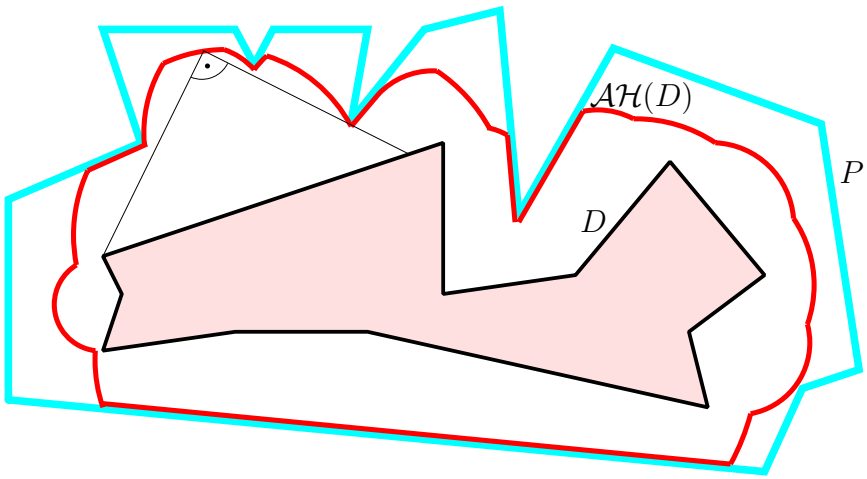

Figure 1: The angle hull $\mathcal{AH}(D)$ with respect to a polygon P.

How long is the photographer's path in terms of the perimeter of her model, D? The following upper bound plays a crucial role in our analysis.

Theorem 1 *Let P be a simple polygon containing a simple polygon D. The arc length of the boundary of the angle hull, $\mathcal{AH}(D)$, with respect to P is less than 2 times the length of D's boundary. This bound is tight.*

Since the angle hull is independent of the robot exploration strategy, and interesting in its own right, we are discussing it separately in Part II of this paper [13]; there we will prove Theorem 1.

The organization of Part I is as follows. Section 2 contains a hierarchical description of the strategy and of its analysis. In Section 2.1 we first discuss how to explore a single corner, that is, a single reflex[2] vertex one of whose adjacent edges has not yet been visible. The robot explores these corners in a sophisticated order: Of all reflex vertices that touch the visible area from the right the robot attempts to explore the one that is clockwise first on the polygon's boundary, as seen from the starting point, s. However, the vertex hereby specified may change as the robot moves. From Theorem 1 we obtain a bound to the length of the resulting path in terms of the length of the shortest path that leads to the final position.

Then, in Section 2.2, we use this technique for efficiently exploring groups of right reflex vertices in clockwise order. The length of the resulting local tour is shown to be bounded by the perimeter of the relative convex hull[3] ($= \mathcal{RCH}$) of certain *base points*, times a constant.

In Section 2.3 we show how the robot recursively detects, and explores, an exhaustive system of groups of right and left reflex vertices. Groups of vertices that are on sufficiently different recursive levels give rise to base point sets whose $\mathcal{RCH}$s are mutually invisible. Therefore, the sum of their hull's perimeters is less than the perimeter of the $\mathcal{RCH}$ of their union. Since all base points are contained in the angle hull of the optimum watchman tour, W_{opt}, the perimeter of their $\mathcal{RCH}$ must be less than the perimeter of the $\mathcal{RCH}$ of $\mathcal{AH}(W_{opt})$, which in turn can only be less or equal to the perimeter of $\mathcal{AH}(W_{opt})$ itself. Now we can apply Theorem 1 a second time and obtain an upper bound to the total length of all local exploration tours in terms of the length of W_{opt}.

2 The strategy and its analysis

Let P be a simple polygon and let s be a point on its boundary. The *shortest path tree* of s consists of all shortest paths from s to the vertices of P. Its internal nodes are reflex vertices of P. Those vertices touching a shortest path from the right are

[2] A *reflex vertex* is one whose internal angle exceeds 180°.

[3] The *relative convex hull*, $\mathcal{RCH}(D)$, of a subset D of a polygon P is the smallest subset of P that contains D and, for any two points of D, the shortest path in P connecting them.

called *right reflex vertices*, left reflex vertices are defined accordingly. If we follow the shortest path from s to reflex vertex v, one of its adjacent polygon edges remains invisible until v is actually reached. The extension into the polygon of this invisible edge is called a *cut* of P with respect to s.

Exploring a polygon P is equivalent to visiting all of its cuts with respect to the start point s. Figure 2 shows an example of the *optimum watchman tour*, W_{opt}, containing a boundary point, s. Tan and Hirata [20] have provided an off-line algorithm for computing W_{opt} within time $O(n^2)$, for a polygon of n edges.

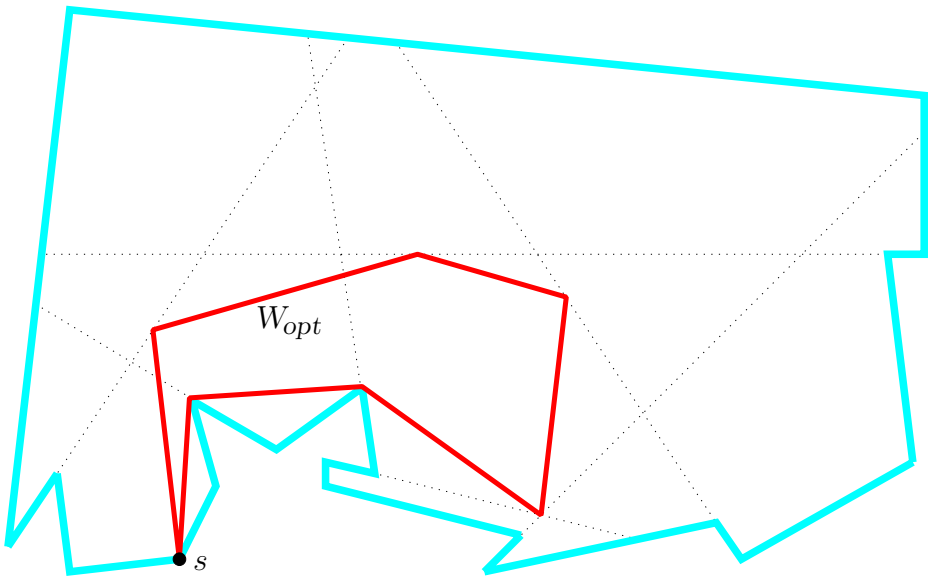

Figure 2: The optimum watchman tour visits all cuts of the polygon.

We say a vertex has been *discovered* after it has been visible at least once from the robot's current position. A vertex is *unexplored* as long as its cut has not been reached, and *fully explored* thereafter.

In an unknown polygon, even exploring a single reflex vertex requires a little care. For example, one cannot afford to go straight to the vertex in order to get to its cut: The cut could be passing by the start point very closely, so that a much shorter path would be optimal.

We avoid this difficulty as follows. Whenever the robot wants to explore a right reflex vertex, r, visible from some local start point, p, it approaches r along the clockwise oriented circle spanned[4] by p and by r, denoted by $circ\,(p, r)$. Consequently, when the robot reaches the cut of r at some point c, the ratio of the length of the circular arc from p to c over their euclidean distance is bounded by $\frac{\pi}{2} \approx 1.57$.

[4]By the circle *spanned* by two points, a and b, we mean the smallest circle that contains these points.

One might wonder if the subproblem of exploring a single vertex can be solved more efficiently by using curves other than circular arcs. This is, in fact, the case; Icking et al. [14] have shown that an optimum ratio of ≈ 1.212 is achieved by curves that result from solving certain differential equations. However, these curves are lacking a useful property possessed by circular arcs: The intersection point, c, of the circular arc with the cut is just the point on the cut closest to p, due to Thales' theorem. This property turns out to be very helpful in our analysis.

In rectilinear polygons, the cut of each visible reflex vertex is known, and two cuts can cross only perpendicularly. This makes it possible to apply a simple greedy exploration strategy: The robot always walks to the cut of the next reflex vertex in clockwise order one of whose edges is invisible; see Deng et al. [9].

For general polygons, this greedy approach is bound to fail, as Figure 3 illustrates. The example polygon shown there suggests exploring left and right reflex vertices separately. However, it is not really obvious how to do this in general, since e. g. the existence of a left reflex vertex at the end of a long chain of right vertices is initially not known to the robot. Therefore, it seems necessary to partition left and right reflex vertices into compact groups that can be explored one by one.

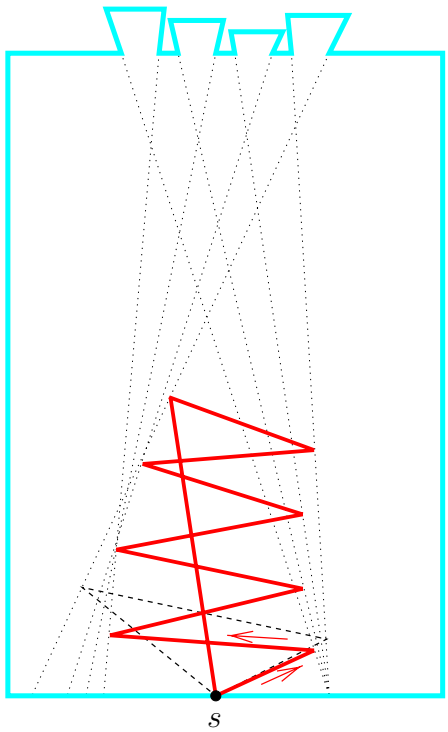

Figure 3: Visiting cuts in the order in which their vertices appear on the boundary does not lead to a competitive strategy.

2.1 Exploring a single vertex

The essential subtask of the robot's strategy is in exploring a single vertex. This is handled by the following procedure *ExploreRightVertex*.

procedure *ExploreRightVertex* (**inout** *TargetList*, **inout** *ToDoList*);

BasePoint := *CP*;

Target := *First* (*TargetList*);

if *Target* not visible **then**
walk on shortest path from *BasePoint* to *Target*
until *Target* becomes visible;

Back := last vertex before *CP* on shortest path from *BasePoint* to *CP*;

walk clockwise along *circ* (*Back*, *Target*)
while maintaining *TargetList* and *ToDoList*

whenever *First* (*TargetList*) changes let *Target* := *First* (*TargetList*);
whenever *Back* becomes invisible update *Back*;

exceptions for walking along the circle:
if the boundary of P blocks the walk on the current circle **then**
walk clockwise along the boundary
until the circular walk is again possible;
if *Target* is becoming invisible **then**
walk towards *Target*
until the blocking vertex is reached;

until *Target* is fully explored;

end *ExploreRightVertex*;

Procedure *ExploreRightVertex* works on two lists of vertices, *TargetList* and *ToDoList*. On entry, *TargetList* contains a list of right vertices, sorted in clockwise order along the boundary, that have already been discovered but not yet explored. When *ExploreRightVertex* is called for the very first time, *TargetList* contains exactly those right vertices that are visible from the start point, s, and have an invisible edge. *ToDoList* can be thought of as a long-term agenda that is passed to *ExploreRightVertex*; however, this procedure will only add to this list but not carry out one of the tasks listed.

We are using the abbreviation *CP* to denote the robot's current position. In our pseudocode, *CP* is a global variable whose value can be changed only by **walk** statements. The robot's current position on calling *ExploreRightVertex* is called a *base point*.

The robot wants to explore the first vertex, *Target*, of *TargetList*. This vertex may have been discovered at an earlier stage, so that it may no longer be visible from the current position. In this case, the robot walks along the shortest path towards *Target* until it becomes visible again; note that this shortest path is in fact known to the robot.

Now the robot starts approaching *Target* along the circular arc spanned by the base point and by *Target*. On the way, a new right vertex, r, may be discovered. If one of its edges is invisible, r gets inserted into *TargetList*, provided that a certain criterion is met. Namely, the shortest path from the current *stage point*—a vertex defined one level up in the strategy—to r must not contain left turns. A right vertex that violates this criterion is ignored for now. A precise definition of a stage point is given in Section 2.2.

It may happen that the vertex r newly discovered and inserted into *TargetList* comes before *Target* in clockwise order. In this case the robot ceases approaching its old target and starts exploring, from its current position, vertex r. This way, the vertex *Target* currently under exploration may repeatedly change.

It may also happen that the robot loses sight of the base point from which the current execution of procedure *ExploreRightVertex* has started. In this case, the exploration of the current *Target* no longer proceeds along the circle spanned by the base point and by *Target*; instead, it switches to the circle spanned by *Target* and by the last vertex on the shortest path from the base point to the current position, excluding *CP* itself. In the code, this vertex is named *Back*.

If the robot crosses the cut of a right vertex different from *Target* the former vertex is removed from *TargetList* because it has been explored by the way. Eventually, the target itself is deleted from the list when its cut has been reached.

When a right reflex vertex is explored, all of its children in the shortest path tree have already been discovered. Those right vertices having a left child are inserted into *ToDoList*, as candidates for future stages, together with references to their left children.

Finally, there are some exceptional events procedure *ExploreRightVertex* needs to take care of. If the robot's circular exploration path hits the boundary of the polygon, the robot follows the boundary until a circular path again becomes possible. If the robot's view of the target vertex is about to be blocked, the robot walks straight to the blocking vertex and continues from there on a circular path.

For ease of reference we summarize the rules by which *ExploreRightVertex* proceeds.

1. The current *target* vertex, i. e., the vertex whose cut we are intending to reach at the moment, is always the clockwise first among those right reflex vertices

that have been discovered but not yet fully explored, i. e., the first element of *TargetList*. Only such right vertices can be in *TargetList* whose shortest paths from the stage point make only right turns.

2. To explore a right reflex vertex, r, we follow the clockwise oriented circle spanned by r and by the last vertex before CP on the shortest path from the base point to CP.
3. When the view to the current target vertex gets blocked (or when the boundary is hit) we walk straight towards the target (or follow the boundary) until motion according to rule 2 becomes possible again.

Figure 4 demonstrates how this strategy works. Initially, r_3 is the only right vertex visible; consequently, *TargetList* contains only r_3, and the robot's path begins with a circular arc spanned by s and by r_3. At point a, right vertex r_2 becomes visible. It is situated before r_3 on the boundary; therefore, the robot switches to exploring r_2, according to rule 1. Note that the circle spanned by s and by r_2 is passing through a, too, so that it is in fact possible to apply rule 2 at this point.

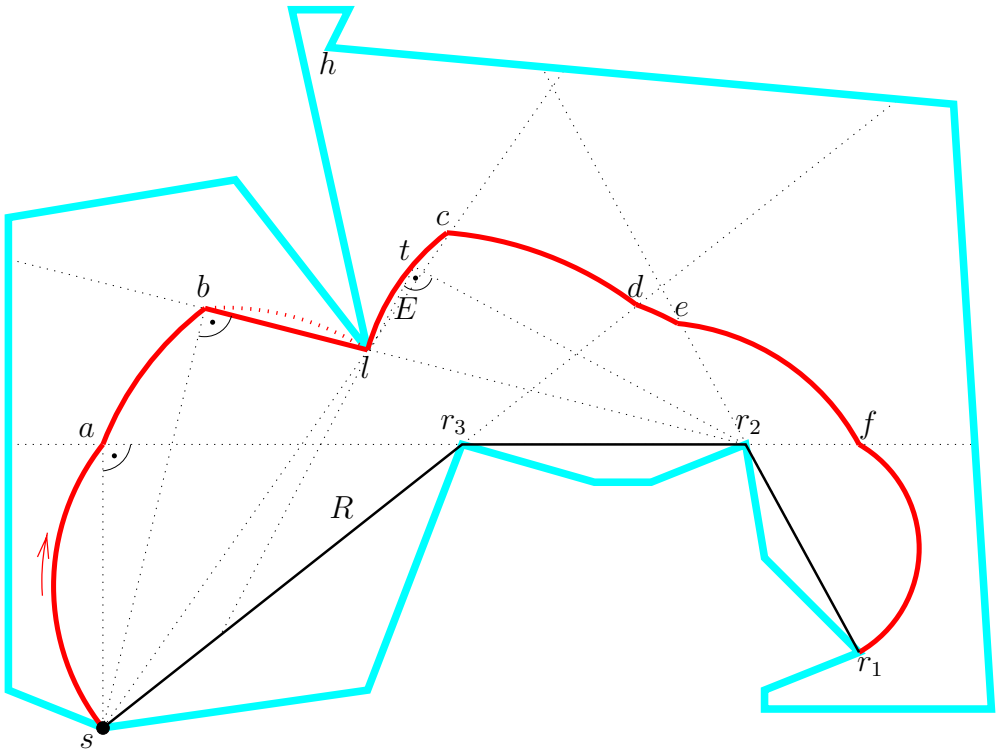

Figure 4: While executing *ExploreRightVertex*, the target vertex is initially r_3, then changes to r_2 and finally to r_1.

At point b, vertex r_2 would become invisible if the robot were to follow the circular arc. But now rule 3 applies, causing the robot to walk straight to the left reflex vertex l. From there, a circular motion is again possible; but the shortest path from s

to CP now contains vertex l. By rule 2, the robot continues its approach to r_2 along the arc spanned by l and by r_2.

Notice that at vertex l, also the right vertex h becomes visible, but it is ignored because its shortest path from s makes a left turn at l.

From c on, the shortest path to s is the line segment. Since the circle spanned by s and by r_2 is passing through c the robot has no difficulties in applying rule 2. At d, the shortest path to s changes again; now it contains vertex r_3. The robot changes its path accordingly and gets to point e from which vertex r_1 becomes visible. From here, the robot explores r_1. Its path follows the circle spanned by r_3 and by r_1, the former changing to r_2 at f. Eventually, the robot arrives at r_1, thereby fully exploring r_1. Here procedure *ExploreRightVertex* terminates.

Now we provide an upper bound to the length of the resulting path. The crucial observation is the following.

Lemma 2 *Suppose procedure ExploreRightVertex terminates with the robot reaching the cut of target vertex r_1 at point c. Then the robot's path is part of the boundary of the angle hull $\mathcal{AH}(R)$ of the shortest path, R, from the base point to r_1, except for straight line segments leading to blocking vertices. Furthermore, point c is the point of the cut closest to the base point.*

Proof. Let t be a point on the robot's path that is not contained on a straight line segment. Assume that, at t, the robot is exploring right reflex vertex r_2, as in the example shown in Figure 4. Since r_2 and r_1 are in convex position relative to the base point, vertex r_2 lies on the shortest path, R, from the base point to r_1.

Now consider the shortest path, T, from the stage point to t. As a consequence of rule 2 and by Thales' theorem, the last line segment, E, of T is perpendicular to the line through t and r_2. Since the backward prolongation of E is bound to hit R, we know that point t can see two points of R at a right angle. Thus, t belongs to the angle hull $\mathcal{AH}(R)$; it lies on the boundary because the angle's sides are both touching path R. □

To estimate the length of the path from the base point to the cut we make use of Theorem 1.

Lemma 3 *The robot's path from the base point to the cut of the target vertex explored by procedure ExploreRightVertex is not longer than twice the length of the shortest path.*

Proof. If the robot's path contains straight line segments leading to blocking vertices, like the segment from b to l in Figure 4, these segments are replaced with circular arcs in the angle hull $\mathcal{AH}(R)$. Thus, the robot's path to the cut of r_1 cannot be longer

than the angle hull's perimeter. If it ends at the point r_1 itself, as in Figure 4, we can apply Theorem 1 to the shortest path as D and obtain the desired upper bound.

If the robot reaches the cut of r_1 at some point different from r_1, we can arrive at the same conclusion using the corollary to Theorem 1 that is stated in Part II of this paper [13]. □

It is important to note that procedure *ExploreRightVertex* ignores such vertices as h in Figure 4, whose shortest paths from the current stage point include left turns. Otherwise, it would not be clear how to apply Lemma 2.

There is a symmetric procedure *ExploreLeftVertex* which is identical to *ExploreRightVertex*, except that left/right, and clockwise/counterclockwise are exchanged.

2.2 Exploring a group of vertices

Each exploration of a group of vertices starts from a *stage point*. The importance of stage points lies in the fact that they are visited by the optimum watchman tour, W_{opt}, too. The first stage point encountered is the robot's start point, s. All stage points are vertices of the shortest path tree of s; the shortest path from s to any vertex of a group leads through the group's stage point.

The exploration of a group of right vertices is performed by procedure *ExploreRightGroup*.

procedure *ExploreRightGroup* (**in** *TargetList*, **out** *ToDoList*);

  StagePoint := *CP*;

  ToDoList := empty list;

  while *TargetList* is not empty **do**
    ExploreRightVertex (*TargetList*, *ToDoList*);
      (* *CP* is now on the cut, C, of the last target. *)
    walk to closest point to *StagePoint* on C
      while maintaining *TargetList* and *ToDoList*;

  walk on the shortest path back to *StagePoint*;

end *ExploreRightGroup*;

The stage point of a right group is always a left vertex. Initially, *ToDoList* is empty, whereas *TargetList* contains a sorted list of unexplored right vertices whose shortest path from the base point makes only right turns. Among them are all unexplored right vertices visible from *StagePoint*.

Roughly, the group exploration proceeds by repeatedly calling procedure *ExploreRightVertex* introduced in Section 2.1 until *TargetList* becomes empty. Afterwards, all right vertices initially present in *TargetList* have been explored, together with their *purely right descendants* in the shortest path tree of s.[5] This set of vertices constitutes a *group*, by definition.

On returning from a call to *ExploreRightVertex* the robot has just explored the clockwise first vertex of *TargetList* and is now situated of this vertex' cut. Before it continues, the robot walks along this cut to the point closest to the stage point; this will be the base point in the next execution of *ExploreRightVertex*. The reason for this step will become clear in the proof of Lemma 6; essentially, it keeps the robot closer to the optimum watchman tour.

Once the last vertex of *TargetList* has been explored, the robot walks back to the stage point, thus completing the exploration of the group. Now *ToDoList* contains, of all right vertices explored, those who have left children, together with references to the latter.

For an example, see Figure 5. Point s is the stage point and also the first base point, and *ExploreRightVertex* is called with $\mathit{First}(\mathit{TargetList}) = r_6$. While exploring r_6, point r_1 is discovered at point a and becomes $\mathit{First}(\mathit{TargetList})$. At $CP = b$ procedure *ExploreRightVertex* returns. Meanwhile, r_2 and r_5 have been added to *TargetList* while r_1 and r_6 have been removed. Point b is also the closest point to s on the current cut.

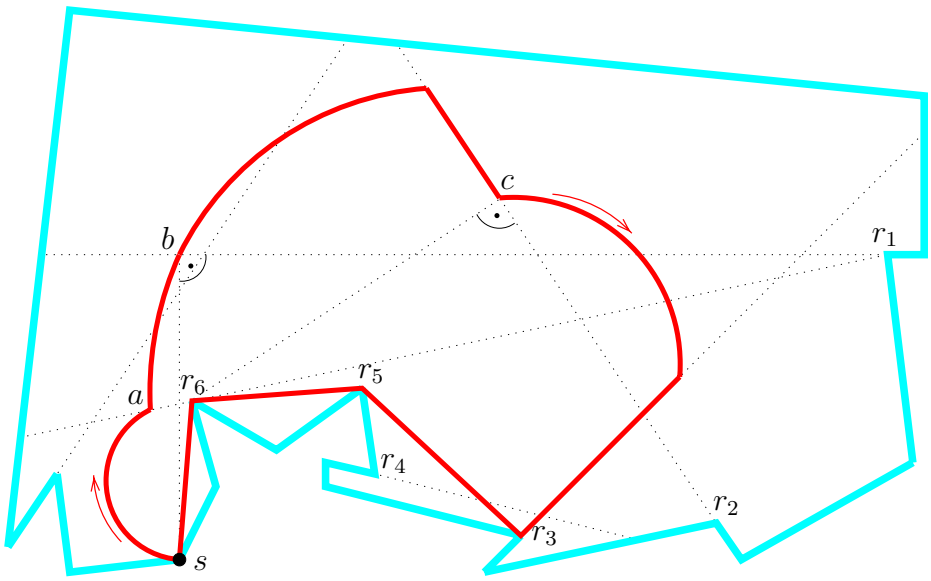

Figure 5: Exploring a group of right vertices.

[5]Vertex w is called a *purely right* descendant of vertex v in the shortest path tree of s if w is a right vertex and if the path from v to w makes only right turns.

As we continue with exploring *First* (*TargetList*) $= r_2$, point r_5 gets explored by the way. Once the cut of r_2 is reached, we walk to c, the closest point to s on the cut. Similar for r_3; while walking along the cut to the point closest to s, which is r_3 itself, r_4 gets explored and no unexplored right vertices remain.

As before with *ExploreRightVertex*, for *ExploreRightGroup* we also have a symmetric counterpart, *ExploreLeftGroup*.

First we prove a useful structural result.

Lemma 4 *Suppose that procedure ExploreRightGroup generates the base points* $b_1, \ldots, b_m$ *in* m *consecutive calls of subroutine ExploreRightVertex. Then the shortest paths from the stage point to* $b_1, \ldots, b_m$ *are in clockwise order.*

Proof. Let base point b_i be situated on the cut of right reflex vertex v_i. Since each call to *ExploreRightVertex* explores the clockwise first right vertex that is still unexplored, $v_1, \ldots, v_m$ appear in clockwise order on the boundary. The stage point must be situated below the cuts of v_i and v_{i+1} as these are unexplored right vertices. The same holds for the last point, p, the shortest paths from the stage point to b_i and b_{i+1} have in common. Moreover, b_i must be below the cut of v_{i+1} because the latter is still unexplored when the robot reaches b_i; see Figure 6. Since neither of the shortest paths nor the cut between v_i and b_i can be penetrated by the polygon's boundary, the claim follows. □

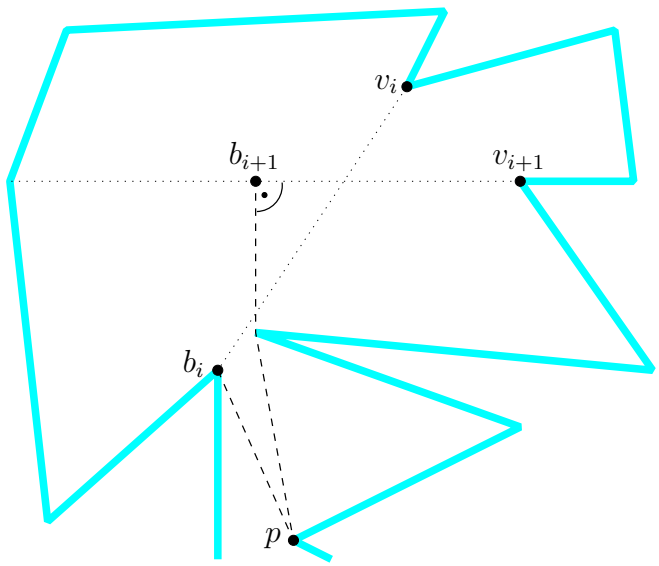

Figure 6: As seen from p, the shortest path to b_i runs to the left of the shortest path to b_{i+1}.

Now we turn to analyzing the length of the path the robot spends on exploring a group of vertices.

Lemma 5 *The robot's path between two consecutive base points is at most* 3 *times as long as the shortest path.*

Proof. Let us call the base points b_1 and b_2, and let c be the point where $cut(v_2)$ is reached. Then c is also the cut's closest point to b_1, by Lemma 2. By Lemma 3, the robot's path to c is not longer than twice the length of the shortest path from b_1 to c and therefore is also not longer than twice the length of the shortest path from b_1 to b_2, see Figure 7.

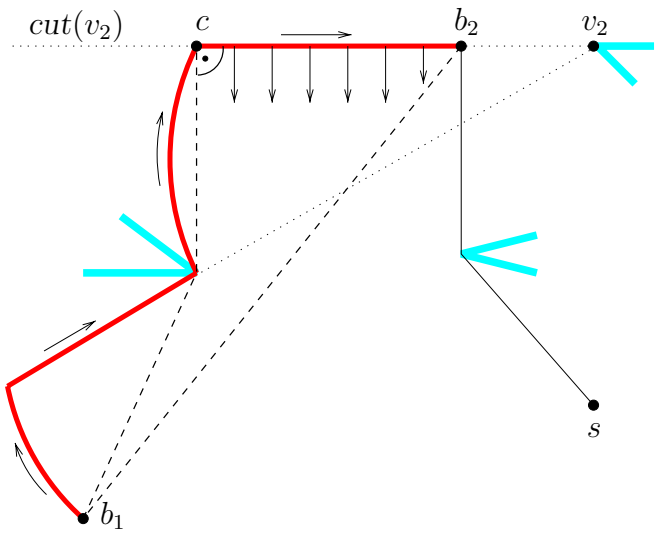

Figure 7: Line segment $c\,b_2$ must be shorter than the shortest path from b_1 to b_2.

It remains to account for the walk along the cut from c to b_2. This line segment can be orthogonally projected onto the shortest path from b_1 to b_2 and, therefore, it must be shorter.

Observe that Figure 7 is in fact generic: As seen from s, the shortest path to b_1 runs to the left to the shortest path to b_2, by Lemma 4; base point b_1 must be located below the cut of v_2; the shortest paths from b_1 to c and from s to b_2 cannot cross because they are both shortest paths to this cut. □

The next steps consist in comparing the length of a *ExploreRightGroup* tour with the relative convex hull of the base points visited.

Lemma 6 *The length of a path caused by a call to ExploreRightGroup does not exceed* $3\sqrt{2}$ *times the perimeter of the relative convex hull of the base points visited.*

Proof. Let sp be the stage point and $sp = b_0, \ldots, b_{m-1}, b_m = sp$ the sequence of base points visited by *ExploreRightGroup*. Due to Lemma 4, $b_1, \ldots, b_{m-1}$ appear in

clockwise order as leaves of the shortest path tree from sp to $b_1, \ldots, b_{m-1}$. So, even factor 3 of Lemma 5 would apply if all base points b_i were vertices of their relative convex hull, $\mathcal{RCH}$, in P.

Suppose that for $i \leq k-2$ the base points b_i and b_k are situated on the boundary of $\mathcal{RCH}$ and the points $b_{i+1}, \ldots, b_{k-1}$ in between are not. The shortest path from sp to the cut of each of them must have a right angle to the cut because of Lemma 2; see Figure 8.

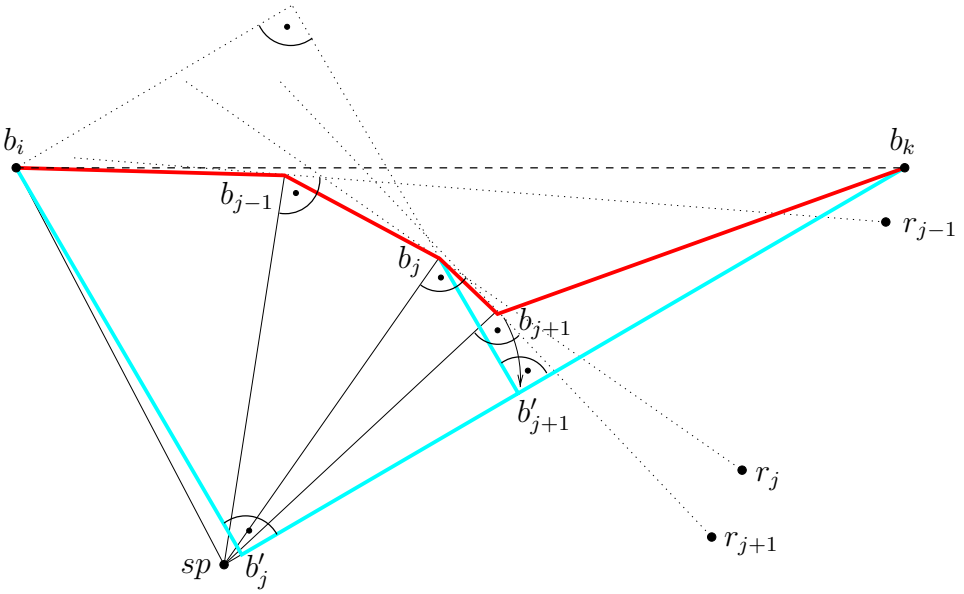

Figure 8: The cut of vertex r_j containing the base point b_j must not intersect the shortest path from sp to b_{j-1}.

For $i < j < k$, if base point b_j were too close to sp then its cut would intersect every possible path from s to b_{j-1}, in particular the robot's path, contradicting the fact that vertex r_j is explored *after* r_{j-1}. Therefore, the cut of b_j must pass above b_{j-1}.

While maintaining these properties, we move the base points one after the other such that the path $b_i, \ldots, b_k$ becomes even longer. For all vertices v of this path, starting with b_{k-1} and going back to b_{i+1}, we do the following. If the path makes a left turn at v, like at b_{j+1} in Figure 8, then we move v to the point on the shortest path from sp to v's successor such that the left turn is a right angle, see b'_{j+1}. Note that every left turn must be an obtuse angle which again is due to the fact that the cut of b_j must pass above b_{j-1}. In case of a right turn we do nothing. Eventually, we end up with a path whose left turns are all right angles.

Now a maximal sequence of right turns, in the example the chain from b_i to b'_{j+1}, can be replaced by one left turn of 90° which is clearly longer than the chain, see b'_j and the rectangle with vertices b_i, b'_j, and b'_{j+1}. Finally, no right turn remains and the new path makes only one left turn of 90° for which the claim is obvious. □

In relating the robot's path to the optimum watchman tour, the following lemma is crucial.

Lemma 7 *All base points are contained in the angle hull $\mathcal{AH}(W_{opt})$.*

Proof. A base point b is, by definition, the closest point to s of a cut. The optimum watchman tour W_{opt} connects the start point s to the cut. Let E be the last edge of the shortest path from s to the cut, i. e. to b, as in procedure *ExploreRightVertex*.

In most cases, edge E is orthogonal to the cut. Then we have a right angle at b whose one side goes along the cut and touches W_{opt}, while the other side K extends edge E. Either the other endpoint of E equals s, so that K touches W_{opt} in s, or K separates s and the cut, and W_{opt} must also be touched by K.

In the remaining case, when there is no right angle between edge E and the cut, point b must be one endpoint of the cut, i. e., it is the target vertex itself or the other endpoint. By similar arguments there is an angle of $> 90°$ both of whose sides touch W_{opt}. □

2.3 Subdividing the polygon

Now we want to combine the exploration of several groups of vertices to finally explore the whole polygon P. This is done by making the *ExploreGroup*-procedures recursive.

procedure *ExploreRightGroupRec* (**in** *TargetList*);

ExploreRightGroup (*TargetList*, *ToDoList*); (* *ToDoList* gets filled in. *)

Clean up *ToDoList*:
retain only those right vertices in *ToDoList*
which are highest up in the shortest path tree;

for all vertices v of *ToDoList* in clockwise order **do**
walk on the shortest path to v; (* connect stage points *)
ExploreLeftGroupRec({all known left descendants of v in counterclw. order});

end *ExploreRightGroupRec*;

The task of *ExploreRightGroupRec* is to explore, from the current position *CP*, all vertices in the input parameter *TargetList* and everything behind.

ExploreRightGroupRec performs in three steps. The *TargetList* is handed over to *ExploreRightGroup*, so *CP* is the new stage point, and after the exploration we are back at this point. We are given a *ToDoList* of candidates for stage points in recursive explorations.

The next step is a necessary cleanup for the *ToDoList*, which contains all purely right descendants of the current stage point which have left children. Some of these

right vertices are descendants of others in this list, they must be removed from the list. Only maximal (highest up) right vertices are retained, these will become stage points in further steps. To each of these future stage points we associate a list of all known left descendants that were referenced in *ExploreRightVertex* and *ExploreRightGroup*.

Finally, the remaining vertices in *ToDoList* are visited in clockwise order, at each vertex procedure *ExploreLeftGroupRec* is called to explore the list of all known left descendants (as *TargetList*) from there. *ExploreLeftGroupRec* is the symmetric counterpart of *ExploreRightGroupRec* with one particularity. In the **for** loop, the vertices in *ToDoList* are also visited in clockwise order. The reason for this will become clear in the proof of Theorem 11.

To conclude the bottom-up presentation of our strategy, we show the main program. Its task is, of course, to explore a given polygon, P, starting at a boundary point, s. First, in a call to the non-recursive *ExploreRightGroup*, the right vertices visible from s are explored. The next target list contains all left children of the right vertices just explored and the left vertices visible from s. All these, and everyting behind, gets explored by a call to the recursive *ExploreLeftGroupRec* with this target list.

procedure *ExplorePolygon* (**in** P, **in** s);

ExploreRightGroup ({clockwise list of all right vertices visible from s}, *ToDoList*);

TargetList := {all left children of the vertices of *ToDoList*};

Add all left vertices visible from s into *TargetList* and sort counterclockwise;

ExploreLeftGroupRec (*TargetList*);

end *ExplorePolygon*;

Each call of *ExploreRightGroup* or *ExploreLeftGroup* generates a set of base points, the first base point of the set is the stage point. For estimating the length of the complete tour, we distribute all these sets into three categories.

The set of base points generated by the call of *ExploreRightGroup* in *ExplorePolygon* belongs to category 0. For the remaining sets, we use their level of recursion to determine their category: the set of base points generated by a call of *ExploreRightGroup* or *ExploreLeftGroup* at total recursion depth i belongs to category (i mod 3).

For example, the very first call of *ExploreLeftGroup* belongs to category 1, and the calls of *ExploreRightGroup* one level deeper belong to category 2. All calls of *ExploreLeftGroup* have an odd level, and all calls of *ExploreRightGroup* an even level.

A key observation is the following.

Lemma 8 *The relative convex hulls of two sets of base points of the same category are mutually invisible, with a possible exception for their stage points.*

Proof. The recursion depths of two sets of base points, B_1 and B_2, of the same category differ by a multiple of 3, possibly 0, as explained above. Let $s_1 \neq s_2$ be the stage points of B_1 resp. B_2. We distinguish two cases depending on the shortest paths from s to s_1 and s_2.

If stage point s_1 is not on the shortest path from s to s_2 and vice versa then let s_0 be the vertex where the shortest paths from s to s_1 and s_2 separate. W. l. o. g. we assume that s_2 is a left vertex and that the clockwise order on the boundary is s_0, s_1, s_2. The left picture in Figure 9 shows such a situation.

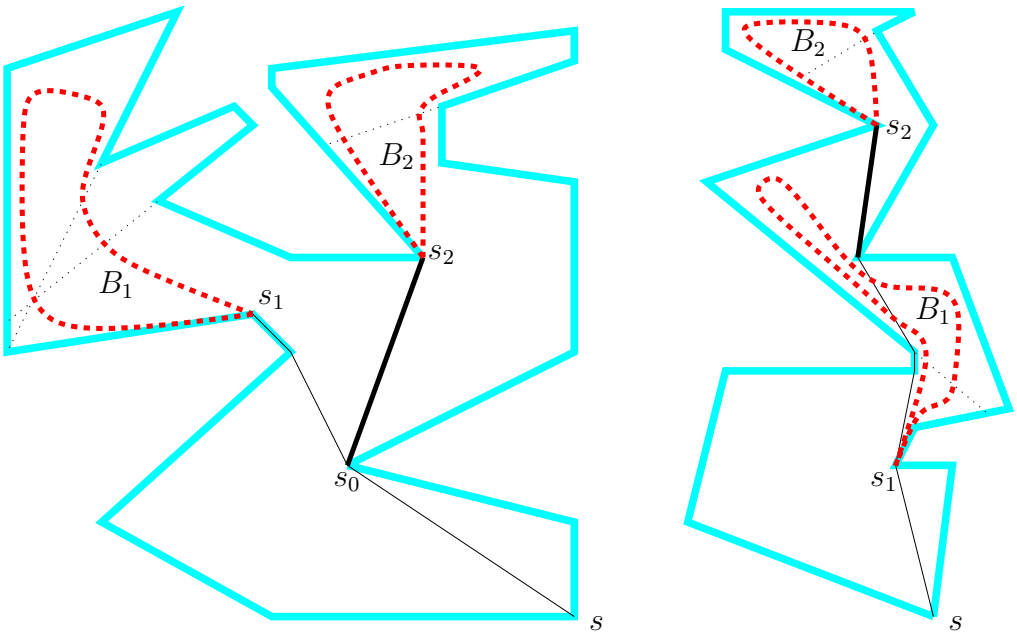

Figure 9: Base points in B_2 can't see B_1, two cases.

The base points of B_2 are on cuts of right vertices whose shortest paths from s_0 all pass through s_2. Therefore, the shortest path from s_0 to s_2 is invisible from any point of B_2, except s_2. But this shortest path separates B_2 from B_1, they are therefore mutually invisible, except for s_1 and s_2. This argument easily extends to the convex hulls as well.

Otherwise assume that s_1 lies on the shortest path from s to s_2. Then the recursion depths of B_1 and B_2 differ by at least three, see the right picture in Figure 9. Similar to the previous case, no point of B_2 can see the shortest path from s_2 to its parent stage point, but this path definitely separates B_1 and B_2.

Note that a difference of three levels is really necessary. If B_2 is only two levels deeper than B_1 then the stage points s_1 and s_2 are of the same type and the parent stage point of s_2 can be a direct descendant of s_1, and therefore it can very well be contained in $\mathcal{RCH}(B_1)$. □

As a consequence, we conclude that the union of all base points of one category has no shorter perimeter than the perimeters of all of its sets of base points together. Let $\mathrm{per}(\mathcal{RCH}(A))$ denote the perimeter of the relative convex hull of set A.

Lemma 9 *Let B_1 and B_2 be two sets of base points of the same category. Then we have $per(\mathcal{RCH}(B_1)) + per(\mathcal{RCH}(B_2)) \leq per(\mathcal{RCH}(B_1 \cup B_2))$.*

As a consequence, we can estimate the path length caused by all calls of *ExploreRightGroup* or *ExploreLeftGroup* in the same category.

Lemma 10 *The path length caused by all calls of ExploreRightGroup and ExploreLeftGroup in one category is less than $6\sqrt{2} \leq 8.5$ times the length of W_{opt}.*

Proof. Let the category consist of sets B_i, $i = 1, \ldots,$ of base points. By Lemma 6, the length of the path created by one call of *ExploreRightGroup* or *ExploreLeftGroup* with set B_i is not greater than $3\sqrt{2}\,\mathrm{per}(\mathcal{RCH}(B_i))$.

The relative convex hulls $\mathcal{RCH}(B_i)$ are mutually invisible (Lemma 8), hence we conclude from Lemma 9 for the path length, L, caused by all calls of *ExploreRightGroup* or *ExploreLeftGroup* of this category

$$L \leq 3\sqrt{2} \sum_i \mathrm{per}(\mathcal{RCH}(B_i)) \leq 3\sqrt{2}\,\mathrm{per}(\mathcal{RCH}(\bigcup_i B_i)).$$

All base points considered are contained in the angle hull of W_{opt}, as Lemma 7 has shown, hence the perimeter of their relative convex hull is shorter than the perimeter of $\mathcal{RCH}(\mathcal{AH}(W_{opt}))$.

The perimeter of $\mathcal{RCH}(\mathcal{AH}(W_{opt}))$ is not longer than the perimeter of the angle hull of W_{opt} itself. By Theorem 1 this is not greater than twice the length of W_{opt} and the claim follows. □

As the main result for our complete strategy, we obtain a factor of 26.5.

Theorem 11 *For a polygon, P, and a start point s on the boundary of P, a procedure call ExplorePolygon (P, s) explores the polygon and returns to s. The total path length used is less than $(18\sqrt{2} + 1) \leq 26.5$ times the length of the optimum watchman tour from s.*

Proof. Since we have three categories of base point sets, all *ExploreRightGroup* and *ExploreLeftGroup* calls together cause a path length of less than $3 \cdot 6\sqrt{2}|\,W_{opt}|$.

It remains to bound the path length caused by the walks during the **for** loops of *ExploreRightGroupRec* and *ExploreLeftGroupRec*. They only connect stage points by shortest paths, and all those stage points are visited in clockwise order along the boundary of P, independently of whether this is done in *ExploreRightGroupRec* or *ExploreLeftGroupRec*.

But all stage points are also visited by W_{opt} in the same order. Therefore, we can be sure that all those walks together make up for an additional path length of at most $|W_{opt}|$. □

3 Conclusions

We have seen that a combination of suitable analysis techniques is necessary for proving an upper bound for the competitive factor of a rather simple strategy. Still, we believe that its actual performance, even in the worst case, is considerably better than the proven bound. Establishing a lower bound for polygon exploration, higher than the trivial $(1 + \sqrt{2})/2 \approx 1.207$, and closing the gap to the upper bound, seem to be challenging problems.

There are many interesting variations and generalizations of the polygon exploration problem. For example, one could study different cost models for the robot's motion. Also, the case of polygons with holes deserves investigation. Here the off-line problem becomes NP-hard, by reduction from the traveling salesperson problem. Recently, Albers et al. [1] have shown that in a rectilinear environment no better competitive factor than $O(\sqrt{k})$ can be achieved for the on-line problem in the presence of k rectilinear holes, what was known before only for general polygons.

References

[1] S. Albers, K. Kursawe, and S. Schuierer. Exploring unknown environments with obstacles. In *Proc. 10th ACM-SIAM Sympos. Discrete Algorithms*, pages 842–843, 1999.

[2] E. M. Arkin, S. P. Fekete, and J. S. B. Mitchell. Approximation algorithms for lawn mowing and milling. Technical report, Mathematisches Institut, Universität zu Köln, 1997.

[3] P. Berman. On-line searching and navigation. In A. Fiat and G. Woeginger, editors, *Competitive Analysis of Algorithms*. Springer-Verlag, 1998.

[4] A. Blum, P. Raghavan, and B. Schieber. Navigating in unfamiliar geometric terrain. *SIAM J. Comput.*, 26(1):110–137, Feb. 1997.

[5] S. Carlsson and H. Jonsson. Computing a shortest watchman path in a simple polygon in polynomial time. In *Proc. 4th Workshop Algorithms Data Struct.*, volume 955 of *Lecture Notes Comput. Sci.*, pages 122–134. Springer-Verlag, 1995.

[6] S. Carlsson, H. Jonsson, and B. J. Nilsson. Finding the shortest watchman route in a simple polygon. In *Proc. 4th Annu. Internat. Sympos. Algorithms Comput.*, volume 762 of *Lecture Notes Comput. Sci.*, pages 58–67. Springer-Verlag, 1993.

[7] W.-P. Chin and S. Ntafos. Shortest watchman routes in simple polygons. *Discrete Comput. Geom.*, 6(1):9–31, 1991.

[8] X. Deng, T. Kameda, and C. Papadimitriou. How to learn an unknown environment. In *Proc. 32nd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 298–303, 1991.

[9] X. Deng, T. Kameda, and C. Papadimitriou. How to learn an unknown environment I: The rectilinear case. *J. ACM*, 45(2):215–245, 1998.

[10] A. Fiat and G. Woeginger, editors. *On-line Algorithms: The State of the Art*, volume 1442 of *Lecture Notes Comput. Sci.* Springer-Verlag, 1998.

[11] F. Hoffmann, C. Icking, R. Klein, and K. Kriegel. A competitive strategy for learning a polygon. In *Proc. 8th ACM-SIAM Sympos. Discrete Algorithms*, pages 166–174, 1997.

[12] F. Hoffmann, C. Icking, R. Klein, and K. Kriegel. The polygon exploration problem: A new strategy and a new analysis technique. In *Robotics: The Algorithmic Perspective, Proc. 3rd Workshop Algorithmic Found. Robot.*, pages 211–222. A. K. Peters, 1998.

[13] F. Hoffmann, C. Icking, R. Klein, and K. Kriegel. The polygon exploration problem II: The angle hull. Technical Report 245, Department of Computer Science, FernUniversität Hagen, Germany, 1998.

[14] C. Icking, R. Klein, and L. Ma. How to look around a corner. In *Proc. 5th Canad. Conf. Comput. Geom.*, pages 443–448, 1993.

[15] J. S. B. Mitchell. Geometric shortest paths and network optimization. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 1999.

[16] S. Ntafos. Watchman routes under limited visibility. In *Proc. 2nd Canad. Conf. Comput. Geom.*, pages 89–92, 1990.

[17] N. S. V. Rao, S. Kareti, W. Shi, and S. S. Iyengar. Robot navigation in unknown terrains: introductory survey of non-heuristic algorithms. Technical Report ORNL/TM-12410, Oak Ridge National Laboratory, 1993.

[18] J.-R. Sack and J. Urrutia, editors. *Handbook of Computational Geometry*. North-Holland, 1999. To appear.

[19] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28:202–208, 1985.

[20] X. Tan and T. Hirata. Constructing shortest watchman routes by divide-and-conquer. In *Proc. 4th Annu. Internat. Sympos. Algorithms Comput.*, volume 762 of *Lecture Notes Comput. Sci.*, pages 68–77. Springer-Verlag, 1993.