Hierarchical Motion Planning Using a Spatial Index

K. Verbarg[1] A. Hensel[1]

Abstract

We investigate the problem of constructing a *shortest path* of a point–like robot between two configurations in the euclidean plane cluttered with (intersecting) convex polygonal obstacles. One common approach is to construct the *visibility graph* and search within this graph in a total time of $\mathcal{O}(n^2)$. We show that in general it is not necessary to construct the entire visibility graph. In contrast, we develop two hierarchical motion–planning techniques based on the *monotonous bisector tree* and the visibility graph, which are shown to be more efficient in scenes of *low object density*. We show that in our setting the visibility graph can be *incrementally* constructed in time $\mathcal{O}(n^2 \log n)$, where n is the complexity of the scene. A shortest path can then be constructed in time $\mathcal{O}(l^4 \log l + n \log l)$, if a shortest path of length l exists. The dependency on n can be further reduced by the use of a spatial data structure complying a query assumption to $\mathcal{O}(l^4 \log l + l^2 \log n)$. If l is not small compared to the diameter D_S of the scene, the algorithm still respects an $\mathcal{O}(n^2(\log n + \log D_S))$ upper bound. If no path exists, then similar estimations for the diameter of a set of obstacles certifying the failure can be stated. We report on experimental results verifying the efficiency of the methods.

Keywords: Shortest path, visibility graph, spatial data structure, computational geometry.

1 Introduction

A fundamental problem in constructing autonomous robots is to compute a collision–free path through a scene of obstacles. Here, we treat the case of finding a *shortest* path between two arbitrary query points in the euclidean plane cluttered with *many* polygonal obstacles. Since we permit intersections of obstacles, we may assume without loss of generality that the obstacles are convex and of constant complexity. Many methods have been proposed to solve this problem — a survey can be found in the textbook of Latombe [1]. The basic concept to solve this special case is the *visibility graph* [2, 3, 4]. It consists of straight–line segments between polygon vertices, which do not cross the interior of any polygon.

The complexity of motion–planning algorithms is usually expressed in terms of the complexity n of the underlying scene. The complexity of a scene of polyhedrons is the number of faces of any dimension. By constructing the visibility graph and applying Dijkstra's algorithm

[1]Department of Computer Science, University of Würzburg, Am Hubland, 97074 Würzburg, Germany, Email: {verbarg,hensel}@informatik.uni-wuerzburg.de, Phone: ++49 – 931 888 5058, FAX: ++49 – 931 888 4600, WWW: http://www-info1.informatik.uni-wuerzburg.de/index_e.html.

we obtain an $\mathcal{O}(n^2)$ shortest path algorithm for non–intersecting obstacles in the plane. There is a strong indication that there is no planar motion planning algorithm (without the demand to compute the *shortest* path) running in time $o(n^2)$ in the algebraic decision tree model, since this problem was shown to be 3Sum–hard [5]. Our idea now is to speed up the construction of a shortest path by *reducing this input complexity* for the motion–planning procedure using a spatial index.

In large scenes, we can observe that the shortest path is often determined by only few obstacles. Therefore, it seems sensible to support the access to locally relevant obstacles with the index so that we do not have to construct the entire visibility graph to find the shortest path. In this paper, we present two different methods which realize this idea: The *ellipse method* and the *segment method*. Our methods can be viewed as *hierarchical* motion–planning techniques, since we do not only use an arbitrary index as a tool for querying objects, but our index is an integral part of the motion–planning procedure. The index we use is the *monotonous bisector tree* [6]. Other attempts to constitute a hierarchical approach are the space–time octree [7], cell decomposition with a binary space partition [8] and hierarchical approximate cell decomposition [9].

To obtain our results we adopt the property of *low object density* [10], which states that an area of constant size may be intersected only by a constant number of objects. The preprocessing is the construction of the monotonous bisector tree (MBT) in time $\mathcal{O}(n \log n)$. When obstacles must be inserted in or deleted from the scene, our data structures can be updated in amortized time $\mathcal{O}(\log^2 n)$. A new obstacle can be inserted in the visibility graph in time $\mathcal{O}(n \log n + k)$, where k is the size of the change. This provides an $\mathcal{O}(n^2 \log n)$ effort to incrementally construct the graph. The ellipse method constructs a shortest path in time $\mathcal{O}(l^4 \log l + n \log l)$, if a shortest path of length l exists. Moreover, if the assumption concerning the query performance of the MBT hold, we obtain an improved running time of $\mathcal{O}(l^4 \log l + l^2 \log n)$. If l is not small compared to the diameter D_S of the scene, the algorithm still respects an $\mathcal{O}(n^2(\log n + \log D_S))$ upper bound. If no path exists, then similar estimations depending on a diameter of a set of obstacles certifying the failure can be stated.

The paper is organized as follows: The next section gives the problem formulation, the model of the scene of obstacles and their properties demanded. Section 3 introduces the monotonous bisector tree and Section 4 the visibility graph with the result on its incremental construction. In Sections 5 and 6 we describe and analyze the two motion–planning methods. Section 7 reports on some experimental results.

2 Problem Formulation

We want to compute a *shortest path* from a start s to a target t in a scene of polygonal obstacles. We could model the obstacles by disjoint simple polygons, but this implies the following disadvantage: Assume, we want to compute a shortest path between s and t in a scene consisting of a simple polygon with complexity $n = 4t + 4$, where t is the number of "teeth" on top of the "comb", as displayed in Figure 1. Using this single obstacle, we would have to take into account all n vertices to compute a shortest path, independent of the length of such a path. Contrary to this, we will model the obstacles by *possibly intersecting convex polygons of constant complexity*, as indicated by the dotted lines in the figure. Then, it is sufficient to compute the shortest path only under consideration of the shaded base of the "comb". Moreover, it is more likely to establish a spatial data structure for the potentially smaller convex parts of constant complexity. On the other hand, this setting will prove to be more complex for the construction of the visibility graph. Fig. 1

Nevertheless, we have to pose some properties of the scene to avoid degenerate cases. To this end we adopt Property 2.2 of [10] (formulated in d–dimensions):

Property 2.1 Let $\mathbb{E}^d$ be a space with a set O of polygonal objects, where each object has a minimal enclosing hypersphere diameter of at least ρ. Then the set O is of *low (object) density* if any region with minimal enclosing hypersphere diameter $D \cdot \rho$, for some constant $D \geq 0$, intersects no more than a constant number, say $\delta_1 (D\rho)^d$, of objects in O.

This property prevents the vertices to admit an accumulation point. This assumption seems to be quite realistic in many practical applications. Van der Stappen showed that e.g. a set of non–intersecting k–fat objects is of low object density.

To be able to construct the visibility graph within sensible time bounds we additionally have to demand another property.

Property 2.2 For a constant $\sigma \in \mathbb{N}$, each object of O intersects at most σ other objects in O. The set of obstacles is then said to be of *low intersection*.

This property does not seem to be a severe restriction for our purposes. If we aim to model scenes of obstacles, as described before, a low intersection of the obstacles with e.g. $\sigma = 8$ (by an orthogonal grid) will be sufficient to cover any simple polygon in the plane. For the sequel we assume that the scenes under consideration fulfill the low object density and low intersection property.

We assume a point–like robot $\mathcal{R}$ which can move in the free space, i.e. $\mathcal{R}$ must not intersect any obstacle. To handle also motions along the boundary of an obstacle $o \in O$, we permit

$\mathcal{R}$ to move on obstacle edges or vertices, *as long as they do not intersect another obstacle.* Therefore, it is not possible to pass a vertex, which is a common vertex of two obstacles.

3 The Monotonous Bisector Tree

Definition

A *spatial index* is a data structure to represent and manage (e.g. under insertion, deletion, changing shape) a large set of spatial objects (the obstacles) to support specific operations. These are first of all nearest neighbor queries, range queries and point location queries, and based thereon visibility queries and motion planning queries. Many indices addressing this problem have been developed, some well-known are the k–d–tree [11], the R–tree [12], the grid–file [13], the quadtree [14] and the Voronoi diagram with Kirkpatrick's index [15].

The spatial index we use in this context is the *monotonous bisector tree* [6]. It is a rooted binary tree, with the objects stored in the leafs. Each internal node contains two artificial and distinct *split values*, which hierarchically partition the set of objects according to a distance function. To be more precise, let O be a finite set of objects, S be the set of potential split values and $d\colon O \times S \to \mathbb{R}_{\geq 0}$ an efficiently computable distance function. Then the monotonous bisector tree $MBT(s, O)$ for an arbitrary initial split value $s \in S$ is a binary tree with the following properties:

$|O| \leq 2$: $MBT(s, O)$ consists of a single node containing the elements of O.

$|O| > 2$: The root r of $MBT(s, O)$ contains two split values s and $s_2 \in S \setminus \{s\}$. They partition O into the two subsets O_1 and O_2, such that the objects in O_1 are nearer to s than to s_2 with respect to the distance d. The objects $o \in O$ with $d(s, o) = d(s_2, o)$ can be assigned to an arbitrary subset. The subtrees of r are then recursively $MBT(s, O_1)$ and $MBT(s_2, O_2)$.

The tuple (s, O) is called *cluster* with *cluster center* s. The *radius* of the cluster is $r(s, O) := \max\{d(s, o) \mid o \in O\}$. Because of the heredity of split values (each split value of a node is kept in one of the two sons) the sequence of cluster radii from the root to a leaf are decreasing. Figure 2 shows the partitioning of six objects in the plane via split values and Figure 3 the related MBT. Fig. 2

Instead of the test $|O| \leq 2$ it has been shown to be more efficient to use a *truncation* Fig. 3 *function*, which interrupts the recursive construction of the tree if the cardinality of O becomes smaller than a certain constant or if the radius of the cluster (s, O) becomes too small.

Construction of the Tree

According to the application that we have in mind, we now consider the d–dimensional euclidean space. Therefore, let $S = \mathbb{E}^d$ and O be a set of n convex polyhedrons in $\mathbb{E}^d$ with *constant* *complexity*. Then our distance function will be computable in constant time (i.e. in $\mathcal{O}(d)$). To construct a tree, we must first choose an initial split value. This can be done in linear time using e.g. approximate center points [16].

The construction of the tree is then performed recursively like the definition states. We only have to determine how the second split value should be chosen. For this purpose two partitioning steps can be applied flexibly. The *balancing step* guarantees that both subsets O_1, O_2 contain at least a linear portion of O. This can be done in linear time and takes care of the balancing of the tree, so that the height of the MBT is bounded logarithmically.

The *contraction step* chooses the split value s_2 such that the cluster radii decrease as fast as possible. Let $o_1, \ldots, o_k \in O$ be the objects with the same maximum distance of s. Let P be a point on any of the o_i, $(i = 1, \ldots, k)$, attaining this distance. Now s_2 is chosen carefully to get as many objects as possible (but at least P) into its cluster. In the worst case only P is separated from the original cluster and the cluster radius does not necessarily decrease. Nevertheless it is possible to reduce the cluster radius to any fraction $q \in]0,1[$ within a fixed number $C(q,d)$ of contraction steps. These steps do not have to be consecutive. In our implementation we used $s_2 := \frac{1}{3}s + \frac{2}{3}P$. The contraction step can be computed in linear time.

Using these partitioning steps, it can be shown that a monotonous bisector tree with logarithmic height can be constructed in time $\mathcal{O}(n \log n)$ and linear space in the worst case. Moreover, the sequence of cluster radii on a path from the root to a leaf can be estimated by a geometrically decreasing sequence. The MBT can be extended to support also dynamic operations in a paging–model describing the access to the slower secondary memory. This extension is the C–tree [17]. It allows insertion and deletion of objects with delayed update of cluster radii in amortized time $\mathcal{O}(\log^2 n)$. If the cluster radii are updated immediately, the amortized time effort for deleting an object is linear in n.

Querying the Tree

The MBT supports many different types of queries, e.g. point queries, range queries and nearest neighbor queries. The way queries are performed is common to all rooted trees. Starting at the root only those sons are explored recursively, whose subtrees are relevant for the query (*pruning of subtrees*). This can easily be checked by examining the cluster of the sons. In a leaf, we must then check each object for satisfying the query separately. The geometrically decreasing cluster radii guarantee an efficient execution of the query, which has been proved by extensive experimental evaluation. Moreover, we hope to obtain non–trivial worst case bounds for certain queries introducing additional demands on the scene like fatness of the objects [18].

To build our motion–planning methods we need two special queries: the *ellipse query* and the *segment query*. They report all objects which intersect a given ellipse in the plane or a given segment, respectively. To this end, we denote by $Q = \mathbb{E}^d$ the set of all possible query points and by d_2 the euclidean metric.

For arbitrary obstacles $o \in O$, split values $s \in S$ and query points $q \in Q$ the following triangle inequalities hold:

$$\forall y \in o\colon d_2(q, y) \le d_2(q, s) + d_2(s, y), \tag{1}$$

$$\forall y \in o\colon d_2(q, s) \le d_2(q, y) + d_2(s, y). \tag{2}$$

For the two related distance functions $d_{\min}, d_{\max}\colon \mathbb{R}^d \times O \to \mathbb{R}_{\ge 0}$

$$d_{\min}(x, o) := \min_{y \in o} d_2(x, y)$$

$$d_{\max}(x, o) := \max_{y \in o} d_2(x, y)$$

we can show with (1) that

$$\begin{aligned}
\forall y \in o\colon d_2(q, y) &\le d_2(q, s) + d_2(s, y) \\
\forall y \in o\colon d_2(q, y) &\le d_2(q, s) + \max_{y \in o} d_2(s, y) \\
&= d_2(q, s) + d_{\max}(s, o) \\
\max_{y \in o} d_2(q, y) &\le d_2(q, s) + d_{\max}(s, o) \\
d_{\max}(q, o) &\le d_2(q, s) + d_{\max}(s, o).
\end{aligned} \tag{3}$$

Similarly, using (2) we obtain:

$$\begin{aligned}
\forall y \in o\colon d_2(q, s) &\le d_2(q, y) + d_2(s, y) \\
\forall y \in o\colon d_2(q, y) &\ge d_2(q, s) - d_2(s, y) \\
\forall y \in o\colon d_2(q, y) &\ge d_2(q, s) - \max_{y \in o} d_2(s, y) \\
&= d_2(q, s) - d_{\max}(s, o) \\
\min_{y \in o} d_2(q, y) &\ge d_2(q, s) - d_{\max}(s, o) \\
d_{\min}(q, o) &\ge d_2(q, s) - d_{\max}(s, o).
\end{aligned} \tag{4}$$

In the construction phase we will now use $d_{\max}$ to determine the partition induced by two split values and to compute the related cluster radii. When performing an *ellipse query*, where the ellipse is determined by its two focal points q_1, q_2 and the diameter D, we can accept the entire cluster of a subtree $MBT(s, O)$ if

$$d_2(q_1, s) + d_2(q_2, s) + 2 \cdot r(s, O) \le D$$

thus with (3) providing for all $o \in O$

$$\begin{aligned} d_2(q_1, s) + d_2(q_2, s) + 2d_{\max}(s, o) &\leq D \\ d_{\max}(q_1, o) + d_{\max}(q_2, o) &\leq D, \end{aligned}$$

and hence all objects $o \in O$ are covered by the ellipse.

On the opposite the complete subtree $MBT(s, O)$ can be pruned if

$$d_2(q_1, s) + d_2(q_2, s) - 2 \cdot r(s, O) \geq D$$

thus with (4) providing for all $o \in O$

$$\begin{aligned} d_2(q_1, s) + d_2(q_2, s) - 2d_{\max}(s, o) &\geq D \\ d_{\min}(q_1, o) + d_{\min}(q_2, o) &\geq D, \end{aligned}$$

and hence all objects $o \in O$ are outside of the ellipse.

For the execution of a *segment query* we can prune a subtree in the same way. But accepting subtrees is not possible, since a segment is not an extended object. Moreover, we are able to extend the MBT such that the queries can be restricted to previously *marked* subtrees and objects. This technique prevents from repeatedly searching the same subtree or objects in subsequential queries, if they have been already mentioned in this context. This is crucial for the performance of the method.

For the estimation of the performance of our motion planning methods we will need a non–trivial time bound for answering region queries with the MBT. Therefore, we conjecture the following generous assumptions in the sequel. They are justified by extensive experimental evaluations [6]. We want to prove these assumptions by using Property 2.1, which inhibit degenerate cases. At this point, you may plug in any tree–based spatial data structure with a given estimation of the query effort and the related results will immediately carry over.

Denote by A_S the area of the smallest sphere covering the scene of obstacles and by A_Q the area of a smallest enclosing sphere of the query region. Let the number of reported objects of the query be q.

Assumption 3.1 The time for performing an ellipse query is bounded by

$$\mathsf{MBT}_{\mathsf{ell}}(A_Q, q) \in \mathcal{O}\left(\frac{A_Q}{A_S} n \log n + q \log n\right).$$

With the above technique of marking already reported objects, we can reduce the effort for multiple queries and state:

Assumption 3.2 The same time bound as in Assumption 3.1 holds, if at most $\mathcal{O}(\log q)$ subsequent queries are done to obtain the same result.

If we restrict ourselves to a scene of fat and non–intersecting objects, then range searching is possible in time $\mathcal{O}(A_Q \log n)$ [19]. Moreover, Assumption 3.2 is fulfilled, because we do not have to query the former regions again.

4 The Visibility Graph

The *endpoint visibility graph* (VG) for disjoint polygons in the plane consists of the vertices of the polygons, which are connected by an edge if the two corresponding vertices can see each other. Two vertices are mutually visible if their supporting straight–line segment is *free*, that means, if it does not cross any obstacle except at its endpoints. Additionally, the segments induced by polygon boundaries are free. The endpoint VG can be generalized to intersecting obstacles, if we consider only *free* vertices of an obstacle, i.e. those which do not intersect any other obstacle. A segment between free vertices is free, if it does not intersect any obstacle, except at its endpoints. Additionally, a polygon boundary is free, if it does not intersect any other obstacle.

In our setting, the problem of finding a shortest path in a scene of (intersecting) polygonal obstacles in the plane can be reduced to searching a subgraph of the endpoint VG, namely the *tangent visibility graph* [2, 3, 4]. It has all free polygon vertices and (in our application) the two (free) query points as nodes and it consists only of free tangents and free obstacle boundaries between two nodes. A straight–line segment is called *tangent* if its supporting line is tangentially touching two obstacles at its endpoints or if one endpoint is a query point and the other is tangentially touching an obstacle.

Recall that the endpoint and the tangent visibility graph may have $\Omega(n^2)$ edges in the worst–case for n objects with a constant number of vertices. Figure 4 displays a tangent VG (edges are solid lines) for three obstacles and two query points. Fig. 4

Most algorithms for visibility graphs are designed (at least with respect to their worst–case bounds) for scenes of n disjoint obstacles. In this respect, the endpoint VG of disjoint segments can be computed in $\mathcal{O}(n^2)$ time [20]. An output–sensitive algorithm running in optimal time $\mathcal{O}(n \log n + m)$, where m is the number of edges in the visibility graph is presented in [21]. A practical $\mathcal{O}(m \log n)$ algorithm is described in [22].

Inserting Obstacles

To build our motion planning methods we need to efficiently *maintain* the visibility graph, when objects are inserted successively. Citing a result of Vegter [23], the endpoint VG for *non–intersecting* obstacles can be dynamically maintained by an expendable algorithm in time $\mathcal{O}(\log^2 n + k \log n)$, where k is number of destroyed and created edges of the graph upon insertion or deletion of a line segment.

It would be sufficient for our purposes to update the tangent VG rather than the endpoint VG. Unfortunately, updating of the tangent VG turns out to be inefficient. But, since both graphs have the same worst-case asymptotic size (quadratic in n), this is not a severe disadvantage. Assume, in the i–th iteration we want to insert the obstacle o into the endpoint visibility graph $\mathcal{G}_i = (V_i, E_i)$. Let $n_i = |V_i|$ and k_i denote the number of segments that have to be deleted from E_i, because they become non–free by inserting o. Our incremental constructions for possibly *intersecting* obstacles runs as follows:

1. Delete all vertices of V_i, which become non–free by the insertion of o, together with their incident edges. This requires $\mathcal{O}(n_i + k_i)$ steps.
2. Insert the vertices and edges of o, as long as they are free. Since we assumed the objects to have constant complexity, this can be done in $\mathcal{O}(n_i)$ time.
3. Test all $\mathcal{O}(n_i)$ segments connecting o with other objects of $\mathcal{G}_i$ and insert those in the VG, which are detected to be free. This results in $\mathcal{O}(n_i \log n_i)$ time effort by sorting the slopes of the tangents and performing a rotational sweep. Additionally, as we will see below, we have to maintain the angular order of incident edges around each vertex. But this can also be realized with the same worst–case time bound.
4. Delete all edges from E_i (their endpoints are all outside of o by Step 1), which become non–free with the insertion of o. This can be done in time $\mathcal{O}(n_i + k_i)$ and is outlined in more detail in the next section.

Since in each iteration the number of edges in $\mathcal{G}_i$ may be quadratic in n_i, we have $k_i \in \mathcal{O}(n_i^2)$ and hence a total update time of $\mathcal{O}(n_i^2)$ providing linear space amount in the size of the VG. But, due to the fact that each potential edge can be inserted or deleted only *once* in the VG when the obstacles are inserted successively, we obtain that the sum of the k_i's over all iterations is $\mathcal{O}(n^2)$. Therefore the time effort to incrementally construct the visibility graph with this update algorithm is $\mathcal{O}(n^2 \log n)$ in the worst–case. We summarize the results in the following theorem.

Theorem 4.1 The insertion of an obstacle with a constant number of vertices in the visibility graph can be performed in time $\mathcal{O}(n \log n + k)$, where n is the number of nodes of the graph and k the number of visibility segments to be deleted by the insertion. The time effort to incrementally construct the entire visibility graph with this procedure is $\mathcal{O}(n^2 \log n)$. The space requirement is bounded by $\mathcal{O}(n^2)$, the worst–case size of the graph. The above results continue to hold, if the obstacles are not restricted to be non–overlapping.

Since we will already have the MBT for the obstacles at hand, we can use the tree to accelerate the insertion of vertices and tangents of obstacle o in Steps 2 and 3. The latter is the bottleneck of the time estimation of the update. Because we obtained all objects to be inserted in the graph be querying the MBT, we can mark them in the tree. We are therefore able to *test* whether a potential new vertex or edge in the visibility graph is free, simply by querying the tree. Practical tests have shown this strategy to improve the performance.

Our motion planning strategies will try to detect those obstacles which might determine the shortest path in question. Our considerations so far show that even if these strategies completely fail, we will do not much worse than the standard approach of computing the VG for *all* obstacles and searching the graph.

Deleting Non–Free Edges

In this section, we will show how to perform the deletion of edges of the visibility graph $\mathcal{G} = (V, E)$ turned non–free by the insertion of an obstacle o. Due to construction, by Step 1 the endpoints of all edges of E are outside of o. Then it suffices to test each of the constant number of bounding segments of o for intersection with edges in E.

According to the suggestion of Welzl and Mitchell [24], we define a *dual graph* $\mathcal{D}$ of $\mathcal{G}$. The dual $\mathcal{D}$ has a node associated with each edge of E. If two edges of E are incident on a common vertex v and are consecutive in the angular order about v, then their corresponding nodes in $\mathcal{D}$ are linked by an edge in $\mathcal{D}$. Therefore, we have $|E|$ nodes in $\mathcal{D}$ and the degree of any node is bounded by 4.

We now consider an arbitrary edge e on the boundary of o. If e is free, then it is an edge in $\mathcal{G}$. Otherwise, by Property 2.2 e intersects at most σ other convex objects in $\mathcal{G}_i$. In this case, we additionally compute during Step 2 the at most 2σ intersection points I of e with the boundary of these objects. Then in Step 3, we compute also the $\mathcal{O}(n_i)$ free segment connecting these intersection points with other objects of $\mathcal{G}_i$ and insert them *only* in the dual graph $\mathcal{D}$.

We color each node of $\mathcal{D}$ *yellow*, if the corresponding edge in E shares an endpoint with e or I, and *red*, if the corresponding edge properly intersects e. The task is to delete all red nodes, i.e. those nodes whose corresponding edge becomes non–free by the insertion of e. We already know many yellow nodes: The edges in $\mathcal{G}$ incident with e are yellow, if e is free. Otherwise, the additionally inserted edges incident with I are yellow.

We are now able to adopt an argument given in [24], which shows that the yellow and red nodes in $\mathcal{D}$ define a subgraph with at most σ connected components. Each connected component contains at least one known yellow node. Hence we can identify all red nodes by a depth first search of this subgraph starting at the known yellow nodes. Let the number of red nodes be at most k. The number of yellow nodes is $\mathcal{O}(\sigma n) = \mathcal{O}(|V|)$. Then the depth first search can be done in time linear of the size of the yellow and red subgraph, that is in $\mathcal{O}(n + k)$, because the degree of any node in $\mathcal{D}$ is at most 4. Note that it is not necessary to construct the dual graph

$\mathcal{D}$ explicitly. Using an appropriate representation of the visibility graph $\mathcal{G}$, compiling also the angular order of edges incident with a vertex, the adjacent (with respect to $\mathcal{D}$) edges of e can be found in constant time, as required.

Searching the Graph

The shortest path can now be found in the visibility graph using e.g. Dijkstra's algorithm [25]. The algorithm runs in $\mathcal{O}(n \log n + m)$ time in the worst–case for a graph with n vertices and m edges.

Since the VG is embedded in the euclidean plane and the length of an edge is the euclidean distance between its endpoints, we can also apply the *euclidean heuristic* [26]. This is essentially an A*–algorithm [27] with the euclidean distance to the target used as an optimistic estimation of the remaining effort. It provides the same worst–case bounds as Dijkstra's algorithm, but was shown to have an average running time of $\mathcal{O}(n)$ for several random graph models. To apply such a model, we would have to introduce more assumptions on the scene of obstacles, to be able to establish statements about the degree of nodes or the VG to be sparse. This seems to be a subject of future research.

It is noteworthy that it suffices to search the tangent VG to obtain a shortest path. But since we need the endpoint VG to be able to perform updates efficiently, we may at least mark the tangents as such while constructing the graph. The search procedure can then be restricted to the tangents. This may improve the performance of the method by a constant factor.

5 The Ellipse Method

In this section we will investigate the question how to efficiently compute a shortest collision–free path between two query points, say start s and target t, in the plane cluttered with obstacles. The idea to reduce the effort is to consider only those obstacles, which are in the sphere of influence of s and t and not those obstacles which are far apart. In large scenes the spatial index provides the efficient retrieval of these locally relevant obstacles, such that the expensive construction of a shortest path can be restricted to these obstacles.

In the *ellipse method* this sphere of influence is specified by ellipses with focal points s and t. The diameter of an ellipse is the sum of the distances of a point on the boundary to its two focal points, which is constant. The method is based on the following observation: If there is a path of length l between s and t, then this path can be found within the ellipse of diameter l. In the sequel d_2 denotes the euclidean distance and $\mathcal{E}_D$ is the ellipse with focal points s, t and the diameter D. For a constant $\alpha \in \mathbb{R}_{>1}$ the method then runs as follows:

1. Let $D := d_2(s, t)$ the diameter of the first ellipse under consideration. We start with an

empty object set and the related empty visibility graph.

2. Query all objects of the scene intersecting the ellipse $\mathcal{E}_D$, which have not yet been reported (that is they are not yet marked) by previous queries of the MBT for this pass of the algorithm.

3. Update the visibility graph by inserting these objects.

4. Construct the shortest path in the visibility graph connecting s and t with respect to the objects under consideration so far. If no such path exists, then terminate the algorithm and report the failure. Otherwise, a path w of length l is found.

5. If w stays within the ellipse $\mathcal{E}_D$, then terminate the procedure and report the shortest path w. If not, we enlarge the diameter of the ellipse to $D := \max\{\alpha \cdot D, l\}$ and repeat the procedure starting with Step 2.

Lemma 5.1 The ellipse method is correct, i.e. a shortest path is reported if one exists or the absence of such a path is detected.

Proof: When the method in Step 4 terminates because no path connecting s and t was found, then there can be no path in the entire scene consisting of a superset of the obstacles considered in the last ellipse of the iteration. This contains the case when s or t are covered by an obstacle. Otherwise the diameters of the ellipses are increasing at least by a factor $\alpha > 1$ until $D \geq l$ and the algorithm terminates. This happens at the latest when the entire scene is covered by an ellipse, and the length of the shortest path found can not increase any more. Upon termination, we have that w is a shortest path staying within the ellipse $\mathcal{E}_D$. Therefore, no obstacle of the scene can intersect w and w is the global shortest path. □

To obtain the following lemma, we make use of Property 2.1.

Lemma 5.2 The lengths $l_1, l_2, \ldots$ of the shortest paths computed by the ellipse method are strictly increasing. The sequence of ellipses considered in the algorithm if a shortest path of length l is found, is $\mathcal{E}_{D_1}, \ldots, \mathcal{E}_{D_m}$, where $D_1 := d_2(s,t)$, $D_{i+1} \geq \alpha \cdot D_i$ and $D_m < \alpha \cdot l$. Thereby, the number of iterations is bounded by $m \leq (\log_\alpha l/\rho) + 4$, where ρ is the lower bound for the minimal enclosing sphere diameter of the objects in Property 2.1.

Proof: Denote by $l_1, \ldots, l_m$ the lengths of the shortest paths computed during the iterations. Since we sequentially insert obstacles, the l_i are increasing. Moreover, these lengths are strictly increasing (except for $l_{m-1} \leq l_m$), because otherwise the algorithm would have terminated earlier in Step 5. If we chose $D_m = l_{m-1}$ in the last iteration, then we have $D_m \leq l_m = l$. In the other case we have $D_m = \alpha D_{m-1} < \alpha l_{m-1} \leq \alpha l_m = \alpha l$, because in iteration $m-1$ the shortest path could not yet be determined.

To estimate the number m of iterations we must ensure that the first ellipse can not be arbitrarily small. To this end, we show that $D_3 \geq \rho$, if $m \geq 3$. We assume that $D_3 < \rho$ (then we also have $D_1, D_2, l_1, l_2 < \rho$) and $m \geq 3$. For each obstacle $o \in O$ under consideration by Property 2.1 we can choose two points $a, b \in o$ with $d_2(a, b) \geq \rho$. A simple argument now shows that the paths w_1 and w_2 can only pass on one side of o, otherwise the lengths l_1 or l_2 would be greater than ρ. For $m \geq 3$, in each of the first two iterations at least one new object is retrieved. On the other hand it is straightforward to see that no object can intersect w_2, because it would have been already reported. We conclude that we then found the shortest path and $m \leq 2$ contradicting the assumption. Notice that we do not have to assume that a path connecting s and t exists to see that $D_3 \geq \rho$.

Hence, we need at most three iterations to achieve that the diameter of the ellipse is greater than ρ. Since the diameter of the ellipses is enlarged at least by a factor $\alpha > 1$ in each iteration, we then do need at most $\lceil \log_\alpha(l/\rho) \rceil$ more iterations, until the diameter is greater than l. The number of iterations if a shortest path exists is therefore $m \leq \lceil \log_\alpha(l/\rho) \rceil + 3$. □

Notice that we do not have to know the value of ρ explicitly when running the algorithm. After three iterations we are already sure that the ellipse is not arbitrarily small anymore.

In the case that *no path between* s *and* t *exists*, the number of iterations depends on the diameter of a set of obstacles giving a certificate of failure. By the same argumentation as in the previous proof, the number of iterations is then logarithmic in this diameter. Hence, the following considerations hold also for the case when no path exists, when the length of the shortest path l is replaced by this diameter.

The immediate consequence of Lemma 5.2 and Property 2.1 is:

Corollary 5.3 If a shortest path of length l exists, the area covered by queries of the ellipse method for some constant $\alpha > 1$ is less than $\alpha^2 l^2$. The number of objects queried by the entire algorithm is bounded by $\delta_1 \alpha^2 l^2$.

Denote by $\mathrm{MBT_{ell}}(A)$ an estimation of the time effort of subsequent ellipse queries regarding a ball of area A and retrieving therefore at most $\delta_1 A$ objects in a scene of low object density. Then we obtain:

Corollary 5.4 In a scene with low object density the time effort of the ellipse method is at most $\mathrm{MBT_{ell}}(\alpha^2 l^2) + \mathcal{O}(l^4 \log l)$, if a shortest path of length l exists. Besides, the space requirement is at most $\mathcal{O}(\min\{n^2, \delta_1^2 \alpha^4 l^4\})$.

Proof: The effort for querying the MBT is estimated by $\mathrm{MBT_{ell}}(\alpha^2 l^2)$ by Corollary 5.3. Now for at most $L := \delta_1 \alpha^2 l^2$ objects the visibility graph must incrementally be constructed in time $\mathcal{O}(L^2 \log L)$. By Lemma 5.2 the number of iterations is in $\mathcal{O}(\log l)$ and therefore the searching effort is at most $\mathcal{O}(\log l)$ times $\mathcal{O}(L^2)$ for the euclidean heuristic. Summing all up results in the time bound of the corollary. The space requirement of the MBT is linear in n and of the VG is quadratic in the number of considered obstacles. □

Throwing in the assumptions which bound the query time of the MBT we finally obtain a time complexity of the ellipse method mainly determined by the length of a shortest path.

Corollary 5.5 With Assumptions 3.1, 3.2 the time effort to compute a shortest path with the ellipse method is $\mathcal{O}(l^2 \log n + l^4 \log l)$.

Proof: With the notions of Assumption 3.1 the area of the entire scene is $A_S \geq n/\delta_1$. We have $A_Q \leq \alpha^2 l^2$ and $q \leq \delta_1 \alpha^2 l^2$. □

But if the length of the shortest path is not small compared to the diameter of the scene, we still have the following worst–case time bound respected by our algorithm. Notice that we do not have to demand Assumptions 3.1 and 3.2 in this context. With D_S we denote the diameter of the scene of obstacles. Notice that in dense scenes [16] or equally distributed scenes the diameter D_S is bounded by $\mathcal{O}(\sqrt{n})$. From the Lemma we conclude that the ellipse method can not behave worse than the standard approach of computing the entire visibility graph and searching it afterwards.

Lemma 5.6 By Property 2.1 the time effort of the ellipse method is bounded by $\mathcal{O}(n^2(\log n + \log D_S))$, regardless whether a path between s and t does exist.

Proof: With the same argument like before at most $\mathcal{O}(\log D_S)$ iterations may occur. In each iteration the tree may be completely traversed in time $\mathcal{O}(n)$. Therefore, retrieving all obstacles by querying the MBT takes at most $\mathcal{O}(n \log D_S)$ time. With Theorem 4.1 we then can construct the entire visibility graph in time $\mathcal{O}(n^2 \log n)$ and search the graph $\mathcal{O}(\log D_S)$ times in $\mathcal{O}(n^2)$. □

If one does not trust in the query performance of the MBT, instead of using the spatial data structure for performing ellipse queries one may alternatively check each obstacle for intersection with the current ellipse. Checking each of the n objects in each of the $\mathcal{O}(\log l)$ iterations leads with Corollary 5.4 to the following observation.

Corollary 5.7 Omitting the monotonous bisector tree, the ellipse method runs in time $\mathcal{O}(\delta_1^2 \alpha^4 l^4 \log l + n \log l)$, if a shortest path of length l exists.

6 The Segment Method

The ellipse method can be further improved if we change the determination of locally relevant obstacles. We will now only query for those objects, which are obstacles of the intermediate shortest path so far. More precisely the segment method does the following:

1. We start with the visibility graph containing the start s and target t as nodes and their connecting straight–line as edge. So far, no obstacles have been considered.

2. Search the VG for the shortest path connecting s and t. If no such path exists then there is no path in the entire scene and the procedure stops. Otherwise a path w is obtained.

3. Query for all objects intersecting w, which have not yet been reported (i.e. are not marked in the MBT). If no such objects exist, then w is a global shortest path. Otherwise insert the objects in the VG and repeat the procedure starting with Step 2.

To avoid testing the same segment repeatedly, we will mark the related edge in the VG after the first use in a segment query. Moreover, we can further improve the method by testing the segments of w subsequently beginning at the start s, only until the first obstacle is found. Then, the shortest path w is already invalid and can be recomputed in a new iteration.

Lemma 6.1 The segment method is correct, i.e. a shortest path is found if one exists or the absence of such a path is detected.

Proof: For the objects under consideration always the shortest path is found. By adding more objects the length of a shortest path can not be reduced. On the other hand upon termination the reported path is collision–free. □

Lemma 6.2 The area covered by queries of the segment method is less than l^2, for l the length of a shortest path. With Property 2.1 the number of obstacles explored by the segment method is at most $\delta_1 l^2$. The space requirement is at most $\mathcal{O}(\min\{n^2, \delta_1^2 l^4\})$.

Proof: In each iteration $i = 1, \ldots, m$ of the algorithm a shortest path of length l_i is computed. The sequence of the l_i is increasing. Therefore, all obstacles queried lie within the ellipse with diameter $l = l_m$. So, for any $\alpha > 1$ the ellipse method considers more objects than the segment method. The result now follows from Corollary 5.3. □

Although we showed that the segment method in general considers less objects than the ellipse method does, we are not able to give improved time bounds like those in the previous section. This is due to the fact that $\Omega(\min\{n, \delta_1 l^2\})$ iterations may occur if only one object is added in each iteration and the entire ellipse $\mathcal{E}_l$ must be searched. An example for $\Omega(l)$ iterations is displayed in Figure 5. The obstacles are rectangles clearly stacked one upon the Fig. 5
other, such that each obstacle must be identified to determine the global shortest path. The dotted lines are the intermediate shortest paths which detect exactly one more obstacle in each iteration.

But, in practice the segment method was clearly superior, as reported in Section 7, although we are not yet able to prove this observation.

7 Experimental Evaluation

Both methods have been implemented and tested in a C++/UNIX–environment. We used the *library of efficient data types and algorithms* (LEDA) [28] of K. Mehlhorn et al. The program is available to the public on the WWW under

http://www-info1.informatik.uni-wuerzburg.de/publications/verbarg/publications.html.

To illustrate the behavior of our methods, an example for the determination of a shortest path with the segment method is displayed in Figure 6. The scene shown in Picture (a) consists Fig. 6
of 100 convex polygons with a minimal enclosing sphere diameter of at most 20% of the diameter of the scene D_S. Moreover, start and target point are marked by a dot. Pictures (b) to (e) present four consecutive iterations of the algorithm. The obstacles intersected by the current shortest path are colored black, the obstacles considered in previous iterations are marked dark and the rest of the scene remains light grey. Finally, in Picture (f) contains the obtained shortest path. Although the length l of the path is even $68\% \cdot D_S$, only 16 obstacles out of a total of 100 of the scene had to be considered for the construction of the visibility graph. Note that the segment method did lookup less obstacles than the ellipse method would. The latter would at least query all obstacles intersecting an ellipse with the diameter l.

To prove the efficiency of the methods and to compare them, we performed extensive test series. The underlying scene displayed in Figure 7 consists of 300 obstacles equally distributed Fig. 7
in the plane. The convex obstacles have at most 8 vertices and their diameter is less than 8% of the diameter of the scene. We performed a total of 1590 shortest path queries answered by both the ellipse (EM) method for the parameter $\alpha = 1.1$ and the segment method (SM). The diagram presented in Figure 8 displays the average number of considered objects divided by the Fig. 8
number of total objects of the scene ("# objects") versus the length of the obtained shortest path divided by the diameter of the scene ("l/D_S"). The segment method shows to be clearly superior over the ellipse method. Moreover both methods seem to be very efficient even when the length of the shortest path is not small compared to the diameter of the scene.

8 Conclusion

We have introduced an approach to reduce the effort to compute a shortest path in the plane by filtering the locally relevant objects with a spatial data structure. Thus, the expensive motion planning algorithm can be restricted to these objects instead of the entire scene of obstacles. We gave tight time bounds and showed both the ellipse method and the segment method to be exact, i.e. a shortest path is found if one exists.

Besides proving the query performance of the monotonous bisector tree for scenes of low object density, we do like to extend our approach to higher dimensions (using a different motion planning strategy) and extended robots in the future. The bottleneck of our methods is the worst–case quadratic size of the visibility graph. Therefore, the question arises, which scenes exhibit a sparse VG. On the other hand, we are currently investigating, whether we can establish an euclidean spanner of the VG, e.g. of linear size. If this spanner can be incrementally constructed, it may be applied directly to our methods.

References

[1] J.-C. Latombe. *Robot Motion Planning.* Kluwer Academic Publishers, 1991.

[2] T. Lozano-Pérez and M. Wesley. An Algorithm for planning collision–free Paths among polyhedral Obstacles. *Communications of the ACM*, **22**(10), 560–570, 1979.

[3] H. Rohnert. Shortest Paths in the Plane with Convex Obstacles. *Information Processing Letters*, **23**, 71–76, 1986.

[4] H. Rohnert. Time and Space Efficient Algorithms for Shortest Paths between Convex Polygons. *Information Processing Letters*, **27**, 175–179, 1988.

[5] A. Gajentaan and M.H. Overmars. On a Class of $\mathcal{O}(n^2)$ Problems in Computational Geometry. Technical Report RUU–CS–93–15, Dept. of Computer Science, University of Utrecht, 1993.

[6] H. Noltemeier, K. Verbarg, and C. Zirkelbach. Monotonous Bisector* Trees – A Tool for Efficient Partitioning of Complex Scenes of Geomtric Objects. In B. Monien and Th. Ottmann, editors, *Data Structures and Efficient Algorithms*, volume 594 of *Lecture Notes in Computer Science*, 186–203. Springer, 1992.

[7] K. Fujimura and H. Samet. Path Planning among Moving Obstacles using Spatial Indexing. In *IEEE International Conference on Robotics and Automation*, 1662–1667, 1988.

[8] C. Ballieux. Motion Planning using Binary Space Partition. Technical Report 93-25, Utrecht University, 1993.

[9] D. Zhu and J.-C. Latombe. New Heuristic Algorithms for Efficient Hierarchical Path Planning. *IEEE Transactions on Robotics and Automation*, **7**(1), 9–20, 1991.

[10] A.F. van der Stappen and M.H. Overmars. Motion Planning in Environments with Low Obstacle Density. Technical Report 33, Utrecht University, 1995.

[11] J.L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, **18**(9), 509–517, 1975.

[12] A. Guttmann. R–Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the ACM SIGMOD*, 47–57, 1984.

[13] J. Nievergelt, H. Hinterberger, and K.C. Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Transactions on Database Systems*, **9**(1), 38–71, 1984.

[14] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison–Wesley, Reading, MA, 1990.

[15] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer, 1987.

[16] K. Verbarg. Approximate Center Points in Dense Point Sets. Technical Report 117, Universität Würzburg, May 1995.

[17] K. Verbarg and H. Noltemeier. Dynamic Environmental Modeling by the C–Tree. In H. Bunke, T. Kanade, and H. Noltemeier, editors, *Modelling and Planning for Sensor–Based Intelligent Robot Systems.* World Scientific, 1995.

[18] M. H. Overmars and A. F. van der Stappen. Range Searching and Point Location among Fat Objects. In J. van Leeuwen, editor, *Algorithms – ESA'94*, LNCS 855, 240–253, Utrecht, NL, September 1994.

[19] A.F. van der Stappen. *Motion Planning amidst Fat Obstacles.* PhD thesis, University of Utrecht, 1994.

[20] E. Welzl. Constructing the Visibility Graph for n–Line Segments in $O(n^2)$ Time. *Information Processing Letters*, **20**, 167–171, 1985.

[21] S.K. Gosh and D.M. Mount. An Output–Sensitive Algorithm for Computing Visibility Graphs. *SIAM Journal on Computing*, **20**(5), 888–910, 1991.

[22] M. H. Overmars and E. Welzl. New methods for computing visibility graphs. In *Proc. 4th Annu. ACM Sympos. Comput. Geom.*, 164–171, 1988.

[23] G. Vegter. Dynamically maintaining the visibility graph. In *Proc. 2nd Workshop Algorithms Data Struct.*, volume 519 of *Lecture Notes in Computer Science*, 425–436. Springer-Verlag, 1991.

[24] J.S.B. Mitchell, G. Rote, and G. Woeginger. Minimum–Link Paths Among Obstacles in the Plane. *Algorithmica*, **8**, 431–459, 1992.

[25] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, **1**, 269–271, 1959.

[26] R. Sedgewick and J.S. Vitter. Shortest Paths in Euclidean Graphs. *Algorithmica*, **1**, 31–48, 1986.

[27] N.J. Nilsson. *Principles of Artificial Intelligence*. Springer, Berlin, 1982.

[28] K. Mehlhorn and S. Näher. LEDA, a Library of Efficient Data Types and Algorithms. Technical Report TR A 04/89, FB10, Universität des Saarlandes, Saarbrücken, 1989.

A Figures

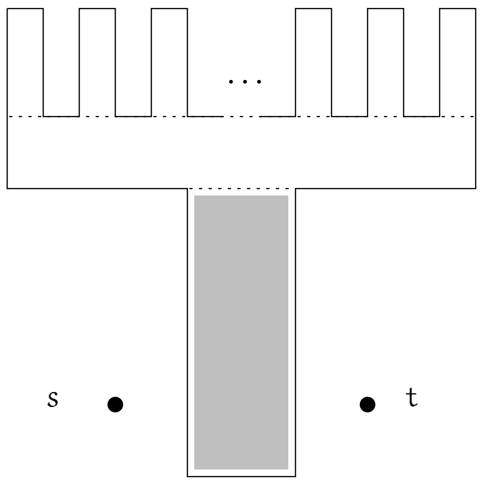

Figure 1: Modeling the obstacles with simple polygons is disadvantageous.

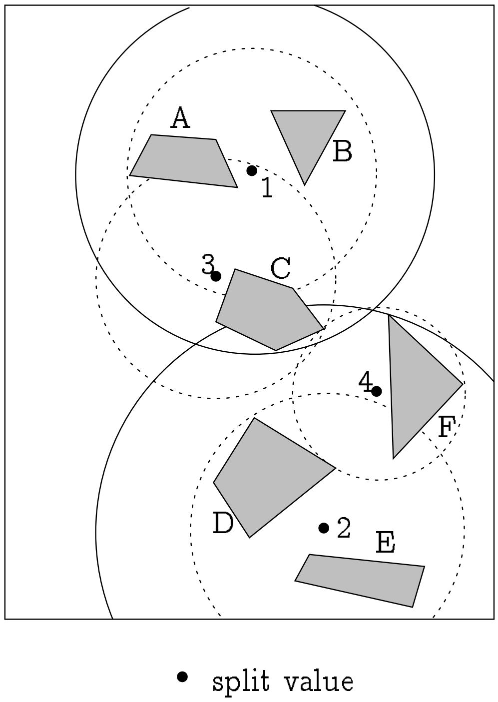

Figure 2: Partitioning of six objects in the plane via split values.

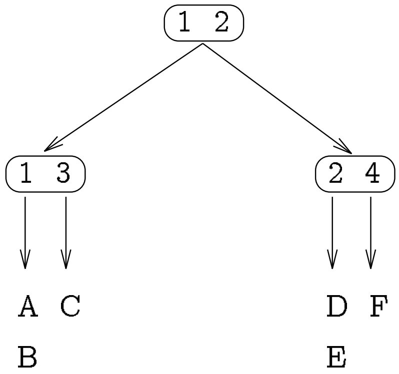

Figure 3: Related monotonous bisector tree for the objects in Figure 1.

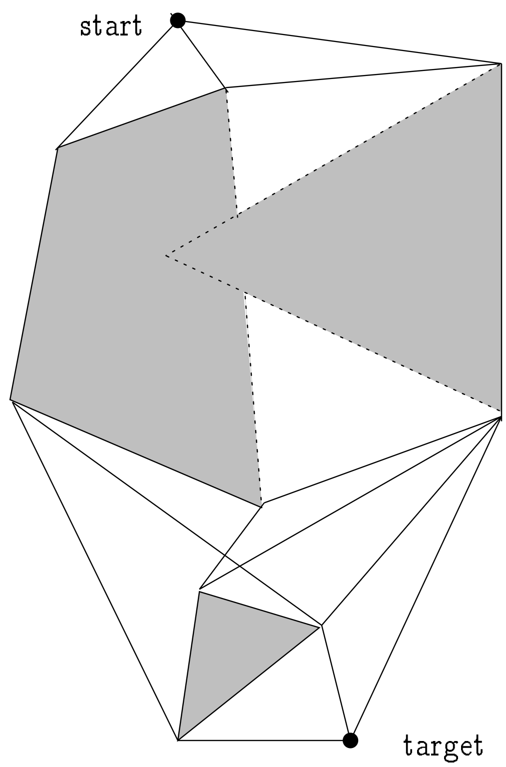

Figure 4: Tangent visibility graph of three obstacles and two query points.

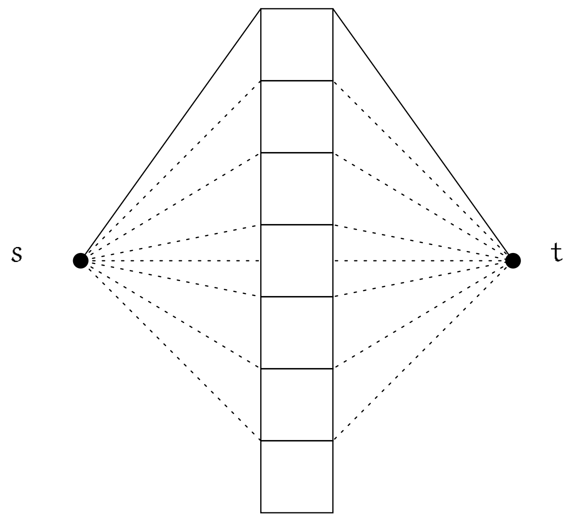

Figure 5: The number of iterations of the segment method can be linear.

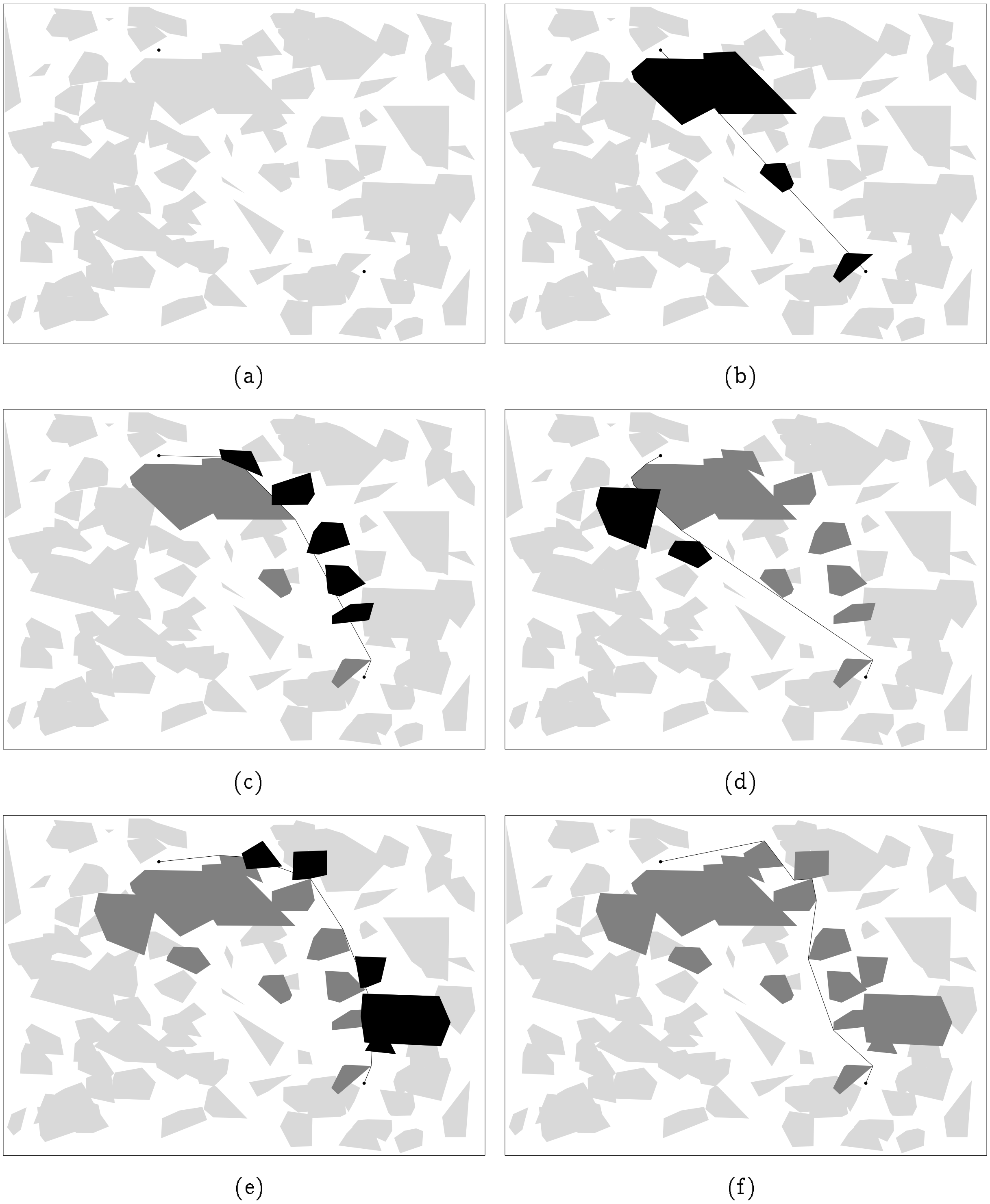

Figure 6: Example of the segment method: (a) shows the scene with start and target point, (b)–(e) the sequence of four iterations and (f) the finally obtained shortest path.

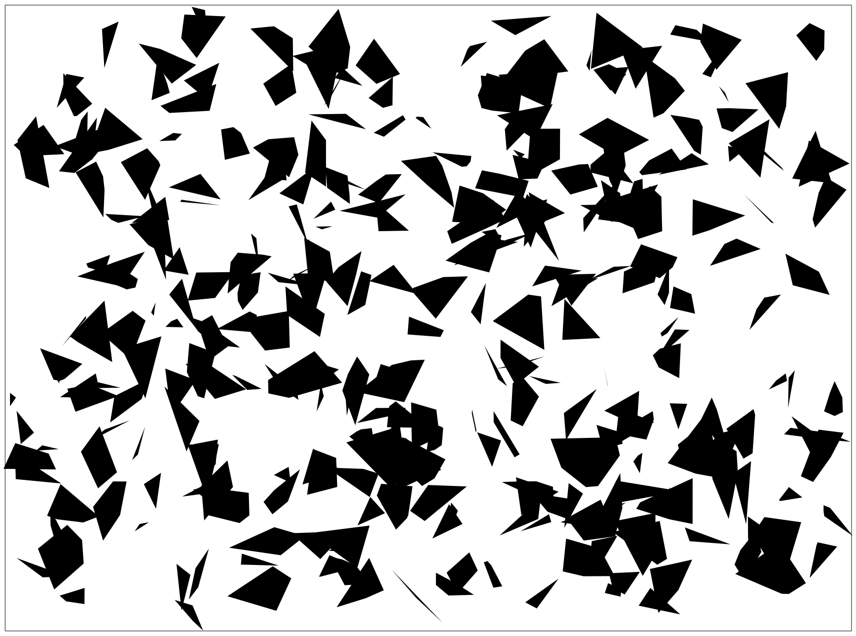

Figure 7: The scene of obstacles used in the test series.

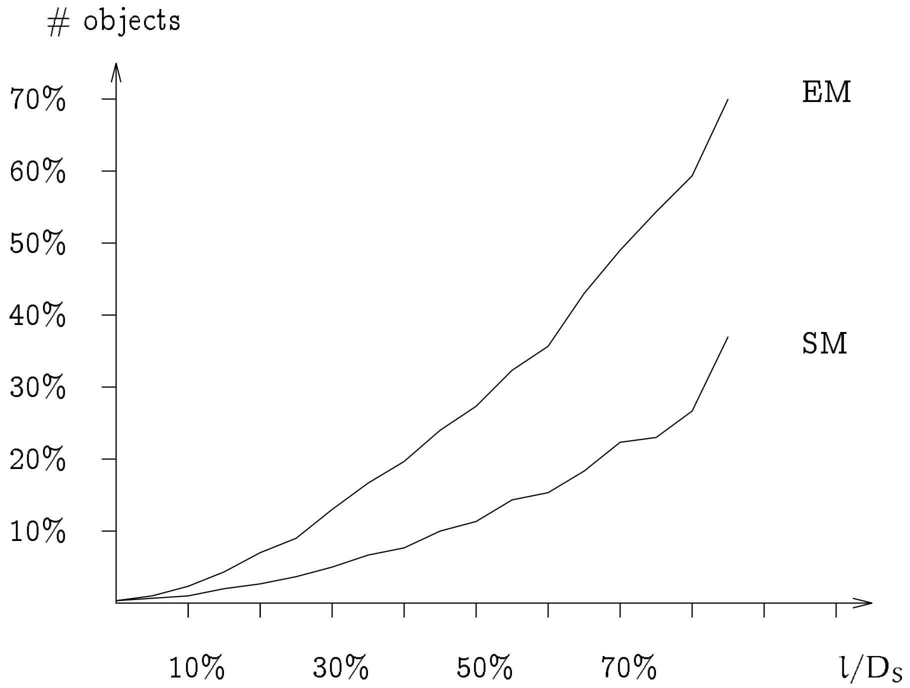

Figure 8: The number of obstacles considered versus the length of the shortest path for the ellipse and the segment method.