We consider three related problems of robot movement in arbitrary dimensions: coverage, search, and navigation. For each problem, a spherical robot is asked to accomplish a motion-related task in an unknown environment whose geometry is learned by the robot during navigation. The robot is assumed to have tactile and global positioning sensors. We view these problems from the perspective of (non-linear) competitiveness as defined by Gabriely and Rimon. We first show that in 3 dimensions and higher, there is no upper bound on competitiveness: every online algorithm can do arbitrarily badly compared to the optimal. We then modify the problems by assuming a fixed clearance parameter. We are able to give optimally competitive algorithms under this assumption. We show that these modified problems have polynomial competitiveness in the optimal path length, of degree equal to the dimension.

Multidimensional online motion planning for a spherical robot

Josh Brown Kramer
Department of Mathematics and Computer Science
Illinois Wesleyan University
Bloomington IL 61701

Lucas Sabalka
Department of Mathematical Sciences
Binghamton University
Binghamton NY 13902-6000
http://www.math.binghamton.edu/sabalka

1. Introduction

This paper concerns online sensor-based motion problems for spherical robots in an unknown bounded n-dimensional environment. Consider Bob, a spherical mobile robot with radius $r > 0$ at starting point S in a space $X \subseteq \mathbb{R}^n$, where X has finite diameter. Bob is equipped with:

- a tactile sensor for feeling and tracing obstacle boundaries, and
- a precise global positioning sensor, which tells Bob its location using global coordinates on X.

For our tasks, Bob will also be able to remember an amount of information proportional to the size of the space, but a priori Bob has no other knowledge of its surroundings.

For any point $p \in X$ and any fixed position for Bob, Bob is *at* p if p is the location of Bob's center, and Bob *occupies* p if p is within distance r of Bob's center.

The three tasks we consider within this setup are:

- *Cover*: Find a path for Bob to move within X to occupy every point in X that can be occupied, and return to the starting point. We denote this task by $COVER$, or $COVER_n$ if n is known.
- *Search*: Given a target point T with unknown coordinates (which is recognizable on contact), find a path for Bob to move within X from S to T, or halt if no such path exists. We denote this task by $SEARCH$, or $SEARCH_n$ if n is known.
- *Navigate*: Given a target point T with known coordinates, find a path for

Bob to move within X from S to T, or halt if no such path exists. We denote this task by NAV, or NAV_n if n is known.

To refer to one of these three tasks without specifying which, we will write $TASK$, or $TASK_n$ if n is explicit.

Note that we do not (yet) assume any further restrictions; in particular, we allow zero-clearance paths (see Section 4).

The purpose of this paper is first to show that if $n > 2$ then, in a precise sense, there is no efficient algorithm to solve any of these problems. We then show that with a minor modification to the problems, these tasks can be accomplished in an efficient manner, and we give efficient solutions.

Online motion algorithms in general are discussed frequently in robotics and computational geometry, and have been a recent active area of research. There are many possible references to algorithms in this area, to which we only name the most relevant to our purposes. There are many more detailed overviews in the literature[2,6,11,12].

Sensor-based motion planning arises in a number of applications. Examples include area coverage problems like cleaning public places, navigation problems like mail delivery in a city or moving objects in a factory, sample acquisition, and planetary exploration; the Mars rover uses autonomous online navigation algorithms (the Field D^* algorithm[5]).

Results concerning online motion algorithms are discussed in terms of the sensors with which the robots are equipped. Often, but not always, robots are given visual sensors to be able to detect (nearby) objects within a line of sight. However, problems requiring only tactile sensors do occur in situations where vision-based sensors are unrealistic. For instance, navigation is often desired in abstract spaces, like the configuration space of a mechanical arm linkage, in which visual sensors, at least in their most literal interpretation, do not make sense.

The motion problems listed above have been frequently studied, usually in special cases. Some of the earliest work on efficient robot motion is that of Lumelsky and Stepanov[13]. That work resulted in the BUG algorithms, which solve the NAV_2 problem for a (point) robot in the presence of arbitrary obstacles. The BUG1 algorithm, described here in Section 3, was proven to run in time proportional to the lengths of perimeters of obstacles in X. However, in terms of the length of the optimal path, BUG1 is not 'competitive'. The notion of linear competitiveness was introduced by Sleator and Tarjan[15], and was generalized by Gabriely and Rimon[7] to non-linear competitiveness. Roughly, optimal competitiveness of an algorithm means that the path it generates has length, in the worst case, proportional to the optimal worst case length generated among all online navigators over all possible environments. Acheiving optimal competitiveness is a common goal. See the references for several examples[1,9,8]. Papadimitriou and Yannakakis[14] provided the first competitive analysis of the NAV_2 problem in specific instances. More recently, Gabriely and Rimon[7] have given a modification of BUG1, called CBUG (described in Section 3), which

is optimally competitive. Gabriely and Rimon show that CBUG is quadratically competitive; CBUG solves NAV_2 with a path whose length is proportional to the square of the length of the optimal offline path and no other algorithm does better in the worst case. To analyze our algorithms, we use the Gabriely-Rimon definition of competitiveness as presented here in Section 2.

For the $SEARCH_2$ problem, a linearly competitive solution in a number of environments was given by Baeza-Yates et al.[1]

For higher-dimensional motion problems, the A^* and D^* algorithms, used in many guises over the past decades, is a predecessor of our work. Also relevant are the various roadmap algorithms. However, to the authors' knowledge, there is no competitiveness analysis for any of these algorithms in the case where there are not visual sensors.

There are simple spaces of dimension $n \geq 3$, the $COVER_n$ task is actually impossible to solve. For instance, suppose we are in 3 dimensions and the only obstacle is a single cube. At a given time, Bob can only touch one point on the surface of the cube. Thus, over time Bob can only trace a 1-dimensional path on the cube's surface. In particular, Bob cannot cover the entire surface. So some slight modification of $COVER_n$ is necessary.

Interestingly, for $n \geq 3$ the optimal online distance for $SEARCH_n$ and NAV_n can be arbitrarily bad compared to the optimal offline distance. One result from this paper, to be made precise via Theorem 7 and Corollary 1, is:

Theorem 1. *If $n \geq 3$ then every algorithm that solves either NAV_n or $SEARCH_n$ has no upper bound on competitiveness with respect to optimal length.*

Thus, some slight modifications of the $SEARCH_n$ and NAV_n tasks are also necessary. The examples that allow us to prove Theorem 1 have very narrow passageways. Roughly, we modify $TASK_n$ so that we only need to consider passageways with a minimum breathing room, herein called the *clearance parameter* ϵ. We will make this precise in Section 4. The introduction of a clearance parameter for motion planning algorithms is common. For instance, the Probablistic Roadmap Method[10] and its various guises use clearance. We place no other constraints on our spaces X: we do not require the obstacles be rectangles, polygons, convex, etc. Although we modify $TASK_n$, our modifications can be physically negligible, as ϵ can be as small as desired, although decreasing ϵ increases the time complexity of the algorithm.

Under the assumption of a clearance parameter ϵ, we prove:

Theorem 2 (c.f. Theorem 7). *Let ϵ and l_{opt} be positive real numbers and let A be an algorithm that solves NAV_n or $SEARCH_n$. There is an environment where the shortest path with ϵ clearance from S to T has length at most l_{opt}, but the path traveled by A has length at least*

$$\frac{l_{opt}^n}{\kappa^{n-2}(r+\epsilon)},$$

where $\kappa = 2\sqrt{2r\epsilon + \epsilon^2}$.

Note that these results fill in a gap in the literature: NAV_n and $TASK_n$ with clearance parameter ϵ are the first tasks proven to have super-quadratic lower bounds on competitiveness.

We go on to present algorithms solving modified $TASK$. The basic algorithm is called Boxes. It solves all three problems. Boxes is optimally competitive for $COVER$ in certain cases:

Theorem 3 (c.f. Theorems 8 and 11). *The algorithm Boxes solves the modified $COVER_n$ problem. Furthermore, in spaces without bottlenecks (see definition 9), Boxes is optimally competitive. In particular, the length of the path it generates is no more than*

$$cl_{opt} + d,$$

where l_{opt} is the optimal path length, and c and d are constants depending on r, n, and ϵ.

Boxes is not competitive when applied to NAV and $SEARCH$. To make it competitive, we restrict movement to an ellipsoid, and progressively increase the volume of the ellipsoid until a solution is found. Our virtual bounding ellipsoid is a direct generalization of the virtual boundary ellipses of Gabriely and Rimon[7]. The following theorem shows that, up to constants, the run time of CBoxes meets the lower bound given in Theorem 1. Thus CBoxes is optimally competitive.

Theorem 4 (c.f. Theorem 10 and 12). *CBoxes is optimally competitive. In particular, it solves the modified NAV_n and $SEARCH_n$ problems and the path it takes has length at most*

$$cl_{opt}^n + d.$$

Here, l_{opt} is the length of the shortest path from start to target with ϵ clearance, and c and d are constants depending on n and ϵ.

This paper is organized as follows. In Section 2, we define the notion of competitiveness that Gabriely and Rimon use. In Section 3, we describe the CBUG algorithm. In Section 4 we modify the definition of the problems by introducing clearance parameter. In Section 5, we prove Theorem 2 by constructing spaces realizing the given bounds. In Section 6, we define the Boxes and CBoxes algorithms. In Section 7 we prove that the algorithms are correct. In Section 8 we prove that the algorithms are optimally competitive. Finally, in Section 9 we describe a number of ways of improving the execution of the algorithms.

A computer simulation of the algorithms contained herein is available online at `http://www.math.binghamton.edu/sabalka/robotmotion`. Further implementations, including improvement of the existing code and enacting the code on physical robots, are in progress.

The authors would like to thank the referees for extensive and helpful comments and suggestions. The second author would like to thank Elon Rimon and Misha Kapovich for many interesting conversations on this material.

2. Competitiveness

Recall from the Introduction that Bob is a spherical mobile robot with the task of moving in an unknown environment X. We want to discuss how "good" a particular online algorithm is for solving a given task. To do so, we present a notion of *competitiveness* for online algorithms. The definition here is the generalized notion of competitiveness appearing in [7]. This definition allows for an arbitrary functional relationship between an algorithm's performance and the optimal performance, not just the traditional linear dependence.

Let P be a problem, NAV_n for example. An *instance* of P is a situation in which the problem should be solved. For online navigation, the instances are given by tuples (X, S, T, r), where X is the space, S the start point, T the target point, and r the radius. Define $t_{opt}(I)$ to be the optimal time, over all algorithms, to complete instance I. Let A be an algorithm that solves P. We want to bound the time required for A to complete a problem, and we want to bound it in terms of t_{opt}. To that end we introduce the following definitions.

Definition 1. Let A be an algorithm solving a task P. Define $t_A(I)$ to be the total execution time for A on instance I. Define w_A, the *worst case function for* A, by

$$w_A(t) = \max_{I \in P}\{t_A(I) : t_{opt}(I) \leq t\}.$$

Thus $w_A(t)$ tells us the most time A could take on an instance if the best algorithm takes time no more than t. In point of fact, in the navigation problem the space of instances is not finite, so the max should really be a supremum.

Definition 2 (Competitiveness). Let P be a problem and let $g : \mathbb{R} \to \mathbb{R}$ be a function. We say that g is a *universal asymptotic lower bound on competitiveness*, or *universal lower bound* for short, if for every algorithm A solving P, $w_A \in \Omega(g)$. An algorithm A solving P is $O(g)$-*competitive* if $w_A \in O(g)$. We say that A is *optimally competitive* if there is g such that g is a universal lower bound on competitiveness and A is $O(g)$-competitive. In this case we say that that $\Theta(g)$ is the *competitive complexity class* of P.

This definition of competitiveness allows for competitiveness to be quadratic, logarithmic, exponential, etc. For example, an algorithm A being (linearly) competitive in the traditional sense is equivalent to being $O(t)$-competitive, which means $t_A \leq c_1 t_{opt} + c_0$ for constants c_0 and c_1. A linear polynomial clearly gives a universal lower bound for competitiveness, and a linearly competitive algorithm is always optimally competitive.

We now turn to our motion tasks. First, note that Bob's position uniquely determines and is uniquely determined by the coordinates of Bob's center. We will refer to Bob's position as a point via this identification. This allows us to talk about, for instance, Bob traversing a path in X. The total execution time of an algorithm A solving $TASK$ may be broken up into physical travel time and onboard computation time. We will neglect onboard computation time when measuring optimality of our

algorithm. This is a defensible assumption, as physical motion typically takes several orders of magnitude longer than onboard computation. To simplify our analysis, we will assume that Bob always travels at a constant speed. This correlates physical travel time with the length, l_A, of the path Bob travels in X while executing A, and we may replace t_A with l_A in our definitions above. These simplifications allow us to compare our performance with that of an optimal offline algorithm (for which computation time is not an issue) by comparing lengths of paths.

We will see that for dimension $n \geq 3$, any algorithm A that solves NAV_n or $SEARCH_n$ has $w_A(t) = \infty$ for every t. That is, knowing the optimal time gives no information about how long A will take on the problem. In Section 4 we will make a slight change to $TASK_n$ that will alleviate this probem

Before we turn to our modification, we present what is known for the NAV_2 problem, which will serve as motivation for parts of our algorithms.

3. Solving NAV_2: the CBUG Algorithm

Our algorithms build on ideas from an optimally competitive algorithm for the NAV_2 task of navigating unknown 2-dimensional environments, called CBUG[7]. The basic CBUG algorithm is itself a refinement of a classical but non-optimally-competitive algorithm, called BUG1[13]. In this section, we present the BUG1 and CBUG algorithms.

BUG1 is guaranteed to yield a solution - that is, Bob will move from S to T if possible - but it is not competitive. The BUG1 algorithm works as follows:

BUG1(S, T)

While not at T:

- Move directly towards T.
- **If** an obstacle is encountered:
  - Explore the obstacle via clockwise circumnavigation.
  - Move to some point p_{min} on the obstacle closest to T.
  - **If** Bob cannot move directly towards T from p_{min}:
    - **Return** 0; Target unreachable.

Return 1; Target reached

BUG1 runs in time proportional to twice the entire length l_b of the boundaries of (an r-neighborhood of) all obstacles (with an easy modification of the algorithm and slightly more careful analysis, the constants of this bound can be improved[13]). However, l_b can be arbitrarily large, even when l_{opt} is bounded. For example, consider the simple situation where S and T are close together, but separated by an obstacle with large perimeter (see Figure 1). One advantage of BUG1 is that only a finite amount of memory is required: Bob must only remember the points T, p_{min},

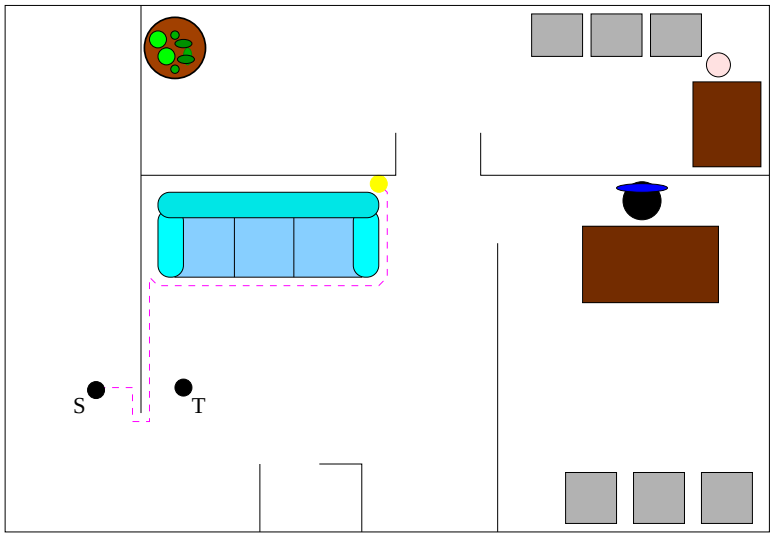


Fig. 1. The first part of BUG1's path is pictured above. BUG1 will traverse the entire perimeter of the room. This is much longer than the optimal offline solution. Instances like this show that BUG1 is not $O(g)$-competitive for any g.

and the first point encountered on the current obstacle.

The CBUG algorithm solves the problem of unbounded competitiveness by introducing a virtual obstacle into the environment. CBUG executes the BUG1 algorithm, but only within an ellipse with foci S and T and of fixed area A_0: Bob treats the ellipse as if it were an obstacle, even though it does not exist. If BUG1 finds no solution within the given ellipse, CBUG repeats the algorithm in ellipses of progressively larger area. See Figure 2.

CBUG(S, T, A_0)

For $i = 0$ to ∞:

- **Execute** BUG1(S, T) within ellipse with foci S and T and area $2^i A_0$.
- **If** Bob is at T:
  - *Return* 1; Target reached.
- **If** Bob did not touch the ellipse while executing BUG1:
  - *Return* 0; Target unreachable.

As the ellipses involved in CBUG are expanding in area, the virtual boundary must eventually contain either a path from S to T or a real obstacle cutting T completely off from S. In the former case, CBUG terminates at T. In the latter

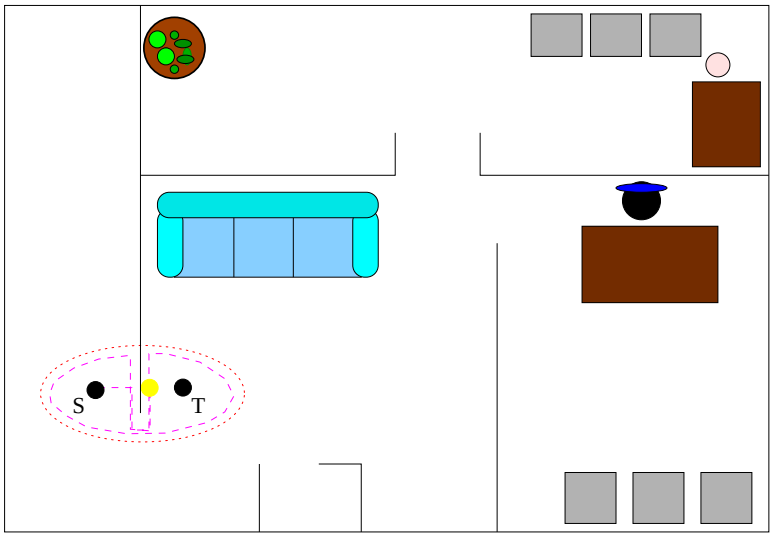


Fig. 2. The dashed line shows the path of a robot executing the BUG1 algorithm within a virtual bounding ellipse from the CBUG algorithm. The ellipse prevents the robot from departing too long from the optimal path.

case, Bob will not touch the virtual boundary, and again CBUG will terminate. Note that CBUG, like BUG1, requires only constant memory: it only need remember the information S, T, A_0, i, and the point p_{min} closest to T on the current obstacle. Usually, we will also have Bob remember the best path from the current point to p_{min}, still requiring only constant memory.

Gabriely and Rimon analyze the competitiveness of CBUG in the following two results:

Theorem 5 (Gabriely and Rimon[7]). *The NAV_2 problem has a quadratic universal lower bound, namely given by*

$$g_r(x) := \frac{4\pi}{6(1+\pi)^2 r}x^2 \sim \frac{.122x^2}{r}.$$

Theorem 6 (Gabriely and Rimon[7]). *If the target T is reachable from S then CBUG solves NAV_2. Furthermore, the length, l, of the path CBUG generates satisfies*

$$l \leq \frac{6\pi}{2r}l_{opt}^2 + dist(S,T) + \frac{6A_0}{2r},$$

where r is the robot's radius. Thus, CBUG is optimally competitive.

Improvements in terms of constants and average-case execution can be made by slightly modifying the algorithm[7].

Gabriely and Rimon mention that the gap between the constants in their lower bound and in the run time for CBug is quite big, and that decreasing this gap is

important. In the next section we will provide a lower bound for competitiveness for $TASK_n$ for general n (see Theorem 7). We note that for large l_{opt}, our lower bound for NAV_2 has a larger constant than the one in Theorem 5. We have therefore narrowed the gap between the upper and lower bounds.

It is not clear to the authors that Theorem 6 is correct. In particular, the proof of this theorem by Gabriely and Rimon[7] uses 3 lemmas and a proposition: Lemmas 4.1-3 and Proposition 4.4. In the proof of Lemma 4.2, the length l of the path that Bob's center traverses is implicitly related to the area A swept out by Bob via the formula $l \leq A/(2r)$, where r is Bob's radius. It is not apparent that this identity holds without further argument. For instance, the r-neighborhood of a fractal curve has finite area, even though a fractal curve has infinite length. The proof that some relationship between l, r, and A holds must at some stage invoke more details of NAV_2 - i.e. that we want the area of a neighborhood of a curve *along the boundary of a neighborhood of an obstacle*, instead of just the area of a neighborhood of a curve. Otherwise, what prevents the path Bob's center follows from being arbitrarily long with respect to the area of its r-neighborhood? In higher dimensions, we can construct spaces which force Bob's center to travel an arbitrarily long distance while covering only a bounded volume. For example, fix some constant k and take $X \subset \mathbb{R}^3$ to be the r-neighborhood of the curve $\gamma(t) = (t, \sin(kt), 0)$, with $0 \leq t \leq 1$, with S and T on the curve at points $\gamma(0)$ and $\gamma(1)$, respectively. Notice that for all k, X is a subset of the box $B = [-r, 1+r] \times [r-1, r+1] \times [-r, r]$. Given the vertical restriction, Bob's center must have height 0. Furthermore, the only points in X with height r are those directly above γ. Thus Bob's center is forced to traverse the entirety of γ. As $k \to \infty$, the length of γ increases without bound, but the volume swept out by Bob is bounded above by the volume of B. There could be such an example for dimension $n = 2$ as well. Note that, *a priori*, such an example could exist even for a very nice environment - i.e. with strong assumptions on the smoothness and fractal dimension of the obstacles.

Using a result of Caraballo[4], the authors can show that $lr \leq cA$ for some (very large) constant c in very restricted settings. A much better result should hold, however, so we do not give our argument here.

We now turn to the problem of modifying the tasks by assuming a clearance parameter.

4. Modifying $TASK_n$: Clearance Parameter

We now wish to analyze the $TASK_n$ problem for arbitrary n. As mentioned in the introduction, there is no optimally competitive algorithm for $TASK_n$, which we will prove in the next section. However, in the process, we will find bounds on competitiveness for slightly weaker problems, defined below, where a clearance parameter is assumed.

We begin by introducing convenient notation to be used throughout the remainder of the paper to discuss the notion of clearance.

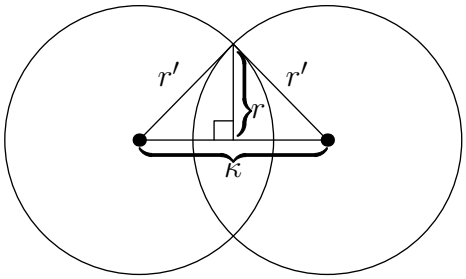


Fig. 3. The quantities r' and κ are defined so that when a sphere of radius $r' = r + \epsilon$ can occupy two points (the two bold points) of distance less than κ apart, then a sphere of radius r can move along the straight line between the two points, by the Pythagorean theorem.

Definition 3 (ρ-neighborhood). Let $Y \subseteq \mathbb{R}^n$ and let $\rho > 0$. Then the *ρ-neighborhood* of Y is the union of all ρ-balls about points in Y:

$$N_\rho(Y) = \{x \in \mathbb{R}^n : \text{ there exists } y \in Y \text{ such that } d(x, y) < \rho\}.$$

Here d is Euclidean distance.

Definition 4 (ρ-path). Let p be a path, and let $\rho > 0$. If the set $N_\rho(p)$ does not intersect an obstacle of X then we call p a *ρ-path* in X.

Definition 5 (κ and r'). Let r be Bob's radius and fix a constant $\epsilon \geq 0$. Define

$$r' = r + \epsilon.$$

and

$$\kappa = 2\sqrt{r'^2 - r^2} = 2\sqrt{2r\epsilon + \epsilon^2}$$

As we see in figure 3, if there are two points of distance at most κ apart such that a sphere of radius r' can occupy either point, then there is r-path between them.

Let $TASK_n$ be one of NAV_n or $SEARCH_n$, and suppose an algorithm A solves $TASK_n$. Let l_0 and ϵ be given constants. In Section 5, we will create a space in which there is an r'-path from S to T of length l_{opt} and the path prescribed by A has length on the order of

$$\frac{l_{opt}^n}{r'\kappa^{n-2}}.$$

In particular, if $n \geq 3$ then the length of path prescribed by A goes to infinity as l_{opt} is fixed and ϵ goes to 0 (see Theorem 7). Thus, no algorithm is competitive with respect to l_{opt} without modifying $TASK_n$:

Definition 6 (Modified NAV_n and $SEARCH_n$). Fix a constant $\epsilon > 0$, called the *clearance parameter*. We define the *ϵ-modified* versions of each task. For convenience, we refer to the ϵ-modified version of $TASK$ as *modified* $TASK$. The *modified* NAV_n

and $SEARCH_n$ problems are to either determine that there is no r'-path from S to T or to find an r-path from S to T (whether or not there is an r'-path).

Note that the path generated by Bob is not required to have ϵ clearance. This is impossible to ask of Bob since he can only navigate by touch. We simply require that *if* there is a path with ϵ clearance then Bob finds *some* path, which only has positive clearance in the unlikely event that Bob encounters no obstacles.

Definition 7 (Modified $COVER_n$). The *modified* $COVER_n$ problem is the $COVER_n$ problem up to ϵ clearance. More precisely, the modified $COVER_n$ problem is to traverse a path such that Bob's center comes within r' of every point that is within r of an r'-path from S.

A slight modification of the example from section 5 shows that even if we insist on the existence of an r'-path, still for every algorithm A it is possible to construct spaces where A takes arbitrarily long in comparison to the shortest r-path. Thus, it is impossible to measure competitiveness with respect to the length of the optimal r-path, even for modified NAV or modified $SEARCH$. For this reason, we modify l_{opt}:

Definition 8 (Modified l_{opt}). For modified $TASK$ we measure competitiveness with respect to the shortest r'-path rather than the shortest r-path. We define l_{opt} to be the length of this shortest path.

From here on, we discuss competitiveness of algorithms solving modified $TASK$ with respect to this modified parameter.

5. Universal Lower Bounds

For the modified $COVER$ problem, there is an obvious linear universal lower bound. In this section we give much stronger explicit universal lower bounds for the modified NAV and $SEARCH$ problems. Note a universal lower bound for modified NAV is automatically a universal lower bound for modified $SEARCH$, as $SEARCH$ is the same problem but with less information. Our universal lower bound will be constructed via spaces where the extra knowledge of the exact location of T does not help Bob, effectively transforming an instance of NAV into an instance of $SEARCH$. Moreover, given an algorithm which solves $SEARCH$, one may always turn an instance of $SEARCH$ into an instance of $COVER$, by moving the target T to the last place a given algorithm searches. We describe spaces such that the optimal offline runtime is proportional to the side length of an n-cube, while an online algorithm runs in time proportional to the n-volume of an n-cube.

The spaces we construct will be 'parallel corridor spaces', $PC(l_0, \epsilon, r, n)$. Each space will consist of a number of floors, and each floor will consist of several corridors of length l_0. These corridors will be squeezed as closely together as possible, overlapping to a great extent but not overlapping so much that Bob can move directly from one to another. We will create an instance of $TASK_n$ by placing S and

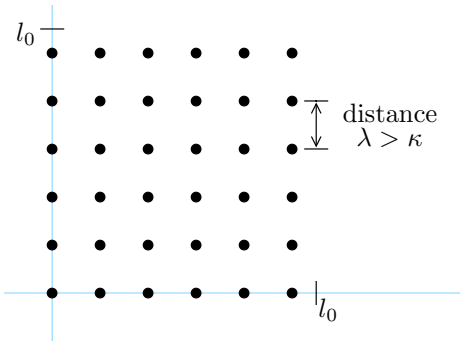


Fig. 4. A sample lattice $L \subset \mathbb{R}^2$, when $n = 4$. Note for this lattice, κ satisfies $6 > l_0/\kappa$.

T at opposite ends of the several corridors, and then blocking all but one corridor. The corridor we choose to leave unblocked depends on the particular algorithm, A, that is being used to solve $TASK_n$. Specifically, the unblocked corridor is chosen to be the last corridor that A visits if *all* of the corridors are blocked. This forces Bob to explore every corridor to get from one side to the other. For examples, see Figures 5 and 6.

For $n = 2$, there is a similarity between our spaces and those of Blum, Raghavan, and Schieber[3], which essentially established a universal lower bound for (a different kind of) competitiveness for the NAV_2 task when all obstacles are polygonal.

5.1. *Constructing the Space*

To construct $PC(l_0, \epsilon, r, n)$, we first construct a single floor. To begin, we construct a finite cubical lattice $L \subset \mathbb{R}^{n-2}$ (see Figure 4). We restrict the coordinates to be elements of the closed interval $[0, l_0]$. We will center corridors of radius r' at these points. Thus we carefully choose the spacing between adjacent points in L to be a bit bigger than κ so that the corridors don't overlap so much that Bob can squeeze between them. We now define this spacing, λ, more precisely.

Denote the number of points that fit into the interval $[0, l_0]$ when spaced distance d apart by $np(d)$. Then $np(d) = \lfloor l_0/d \rfloor + 1$. If $l_0/\kappa \in \mathbb{N}$, then $np(\kappa) = l_0/\kappa + 1$. In this case, choose $\lambda > \kappa$ so that $np(\lambda) = l_0/\kappa$. If $l_0/\kappa \notin \mathbb{N}$ then choose $\lambda > \kappa$ so that $np(\lambda) = np(\kappa) = \lfloor l_0/\kappa \rfloor + 1 > l_0/\kappa$. In either case, the number of points along each side of L is $np(\lambda) > l_0/\kappa$.

Extend L to a subset of $\mathbb{R}^{n-1}$ by attaching an interval of length l_0 to each point in L: $L \times l_0 I \subset \mathbb{R}^{n-1}$, where I is the unit interval. The r'-neighborhood of each of these lines is a *corridor*. Create a series of corridors in $\mathbb{R}^n$ by taking the r' neighborhood of $L \times l_0 I$: set $L' := N_{r'}(L \times l_0 I)$. Figure 6 shows a picture of L' (missing caps on the ends, and with extra black 'flaps') in the case $n = 3$.

Notice that L could have been chosen from a more dense packing (of the $(n-2)$-

cube with side length l_0 by spheres of radius λ) to fit more corridors into L' and thus obtain better constants for the bound (see the Lattice Improvement, Section 9.1). However, this does not affect the competitiveness class of our example.

The set L' is already a collection of parallel corridors, but there are not enough of them. We will stack enough of them so that the height of the stack is roughly l_0. More precisely, stack $h = \lfloor l_0/(2r') \rfloor$ copies of L' on top of one another. That is, place one copy of L' at height 0, one at height $2r'$, one at height $2(2r')$, etc., up to height $h(2r')$. Think of each copy of L' as a floor in a building with h stories.

To be able to access any corridor from any other corridor, add a room at each end of the collection of all corridors so that a robot of radius r' can pass between floors by passing through a room - i.e. both rooms have dimensions roughly $(2r' \times (l_0 + 2r') \times \cdots \times (l_0 + 2r') \times (l' + 2r'))$, where $l' = 2r'h$. Call one room the start room and the other the target room.

At the end of all but one of the corridors, place obstacles that prevent passage from the corridor to the target room. The choice of which corridor to leave open depends on the algorithm, A, that we are building the space for. If all of the corridors were blocked, A would visit every corridor in some order before terminating. Leave the last one unblocked.

We carefully choose the size, shape, and location of the obstacles that are blocking the corridors. Say we wish to block a corridor C, whose axis of symmetry is $x \times l_0 I$, where $x \in \mathbb{R}^{n-2}$. Let P' denote the set of points in C which are as close as possible to but not in the target room. Then P' is a $(n-1)$-ball orthogonal to $x \times l_0 I$ in $\mathbb{R}^n$. Let $P \subset P'$ denote those points whose height (i.e. the value of the last coordinate, which is the coordinate that was increased in the stacking phase, corresponding to the floor) differs from the height of x by g or more, where $g = \sqrt{(r+\epsilon)^2 - ((\kappa+\lambda)/4)^2}$. Then P consists of two connected components of distance $2g$ apart.

Lemma 1. *The obstacle set P chosen is such that:*

(1) obstacles block a robot from exiting a blocked corridor,
(2) obstacles do not block a robot from exiting the chosen unblocked corridor, and
(3) a robot in one corridor cannot sense an obstacle in an adjacent second corridor (and thus determine that it need not go down the second corridor).

Proof. We claim P satisfies the three desired properties, all of which follow from the choice that $\lambda > \kappa$. For, the connected components of P are distance $2g < 2\sqrt{(r+\epsilon)^2 - (\kappa/2)^2} = 2\sqrt{(r+\epsilon)^2 - ((r+\epsilon)^2 - r^2)} = 2r$ apart, which blocks a robot of radius r from passing, so (1) is satisfied. Let C_1 and C_2 be intersecting corridors. Their axes of symmetry are at the same height, z. Notice that the difference, in absolute value, between z and the height of a point in $C_1 \cap C_2$ is at most $\sqrt{(r+\epsilon)^2 - (\lambda/2)^2} < g$. Thus the unblocked corridor has had no obstacles placed in it, so (2) is satisfied. Furthermore, the same calculation shows that a robot can

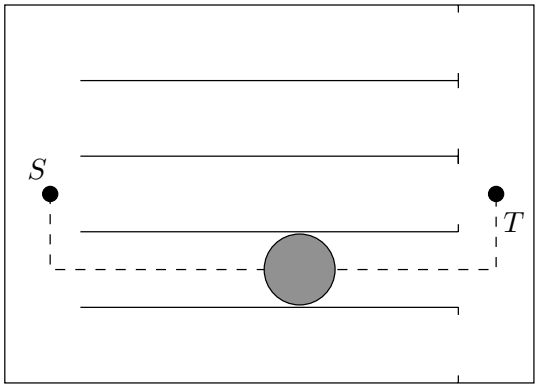


Fig. 5. The space $PC(8r', \epsilon, r, 2)$. The circle represents the robot, and the dotted line indicates the optimal length path from S to T.

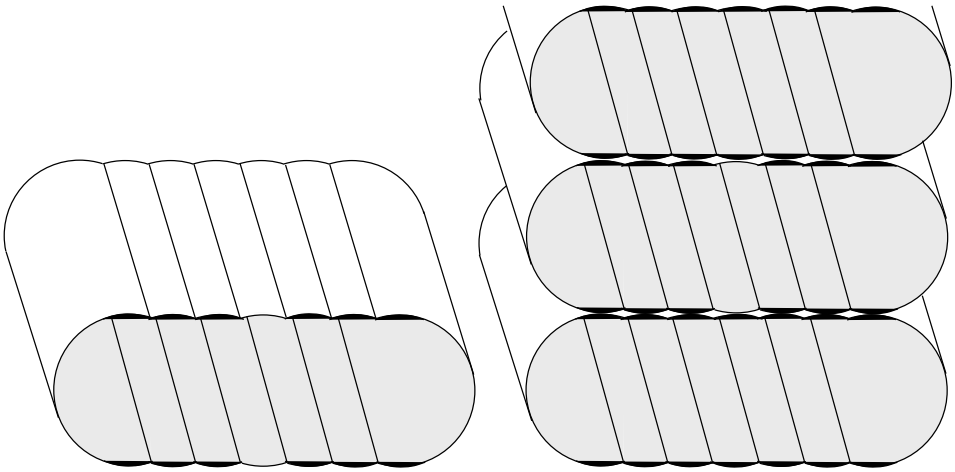


Fig. 6. On the left is a single floor in the case $n = 3$. It has 7 corridors. Notice that exactly one of the them (the middle one) is unblocked; Bob cannot pass through the other corridors because of the black obstacle 'flaps'. On the right is a parallel corridor space with the end rooms removed. It has 21 corridors. Notice that only one corridor is unblocked.

only feel elements of height strictly less than g in adjacent corridors. Thus (3) is satisfied. □

Adding the obstacles P to all but one corridor, we have now finished constructing the space $PC(l_0, \epsilon, r, n)$.

We create an instance of $TASK_n$ by placing the start point S in the center of the start room, and placing the target point T in the center of the target room.

Example 1.

Consider when $n = 2$. In this case, L is a subset of $\mathbb{R}^0$. That is, L is a point. Since L is a point, $L \times l_0 I$ is a line segment of length l_0. If we think of the line segment as sitting horizontally in the Euclidean plane, then taking the r'-neighborhood of the line segment yields: two half-circles, one on the left (concave right) and one on the right (concave left), connected by horizontal line segments of length l_0. For the sake of this example, assume $l_0 = h(2r')$ for some integer h. Then, taking h copies of this corridor and stacking them on top of one another, we have a series of horizontal line segments spaced at distance $2r'$ from each other, with 'caps' on the left and right ends. To create the space $PC(l_0, \epsilon, r, 2)$, on each end we replace the caps with a box of width $2r'$ and height l_0. We declare the left-hand box to be the start box, putting S in its center, and we declare the right-hand box to be the target box, putting T in its center. Finally, we place obstacle 'flaps' in all but one of the corridors on the far right-hand side. See Figure 5.

Example 2.

If $n = 3$ then L is a set of equally spaced points on a line segment of length l_0. The resulting set L' is as shown in Figure 6, plus hemishperical 'caps' on the end. We then remove the caps, and add flaps. In 3D a flap is a disk with a central strip removed. We say that two corridors are *adjacent* if the distance between the corresponding points in L is λ. Notice that if λ were allowed to be smaller than κ then the pinching between adjacent corridors would be less and a robot of radius r could pass directly from one corridor to another without passing through the start room.

5.2. *Analysis of the Space*

Using the parallel corridor spaces we have constructed we prove the following.

Theorem 7. *The modified NAV_n task and the modified $SEARCH_n$ task both have an asymptotic universal lower bound on competitiveness given by*

$$l_{opt}^n.$$

More specifcally, for large enough l_{opt} and for any deterministic navigation algorithm, A, there is a space X with an r'-path from S to T of length at most l_{opt} where the length of the path generate by A has length at least

$$c_n \frac{l_{opt}^n}{\kappa^{n-2} r'},$$

where c_n is a constant depending only on n.

Proof. Consider the parallel corridor space $PC(l_0, \epsilon, r, n)$. One way to get from S to T is to travel from S directly to the unobstructed corridor, travel along the unobstructed corridor to the target room, then travel directly to T. The length l_{opt} is therefore at most the length of a corridor (l_0) plus the maximal distance from S

to a corridor (which is at most $\sqrt{n-1}l_0/2 + r'$) and from a corridor to T (which is also at most $\sqrt{n-1}l_0/2 + r'$). Thus,

$$l_{opt} \leq (1 + \sqrt{n-1})l_0 + 2r',$$

and so

$$l_0 \geq \frac{l_{opt} - 2r'}{1 + \sqrt{n-1}}. \tag{1}$$

By our choice of λ in the construction of the space $PC(l_0, \epsilon, r, n)$, the number of corridors per floor is at least

$$\left(\frac{l_0}{\kappa}\right)^{n-2}.$$

Thus the total number of corridors is at least

$$\left(\frac{l_0}{\kappa}\right)^{n-2} \left\lfloor \frac{l_0}{2r'} \right\rfloor.$$

We point out that to get between adjacent corridors, Bob's center must pass through the start room. This simply follows from the fact that the corridors are spaced more than κ apart and hence the height of the aperture between corridors is less than $2r$ (see Figure 3).

Let A be an algorithm solving $TASK_n$. Initially, block all of the corridors with flaps. As we showed in Lemma 1, Bob cannot determine that a corridor is blocked other than by traveling down that corridor. Thus A has Bob travel all the way down each corridor to determine that each corridor is blocked. A will visit some corridor last. Choose this to be the unblocked one. Then Bob visits each and every blocked corridor. For each blocked corridor, Bob thus traverses a distance of at least $2(l_0 - r')$. Bob travels the unblocked corridor at least once. Thus, any online algorithm in $PC(l_0, \epsilon, r, n)$ will have path length at least

$$2\left(\left(\frac{l_0}{\kappa}\right)^{n-2} \left\lfloor \frac{l_0}{2r'} \right\rfloor - 1\right)(l_0 - r') + l_0. \tag{2}$$

Combining (1) and (2), we have the desired result. □

We note some things. First, together with the additional mild assumption that $\epsilon < r$, 'large enough' depends solely on n and r. Second, even if the algorithm is non-deterministic, there is a non-zero probability that it will take the same sequence of corridors as it did when the decision was made about which corridor to leave open. Thus, this worst case analysis applies even to non-deterministic algorithms. Third, we note the particular case $n = 2$. Letting $\epsilon \to 0$, we see that this distance is quadratic in l_{opt} with a leading coefficient of $1/(4r) = .25/r$. Thus for large l_{opt}, our example forces paths more than twice as long as those forced by the examples of Gabriely and Rimon (see Theorem 6) who mention the importance of closing the gap between the constants in the lower and upper bounds [7] . Finally, when $n \geq 3$,

by allowing ϵ to go to 0 we obtain greater and greater lower bounds on online path length independent of l_{opt}. Thus,

Corollary 1. *If $n \geq 3$ then every algorithm that solves the (unmodified) NAV_n problem or the (unmodified) $SEARCH_n$ problem is not $O(f)$-competitive for any $f : \mathbb{R} \to \mathbb{R}$.*

6. The Algorithms

In this section, we present the Boxes algorithm for solving $COVER$, NAV, and $SEARCH$. We then modify Boxes to get CBoxes, an optimally competitive algorithm for solving NAV and $SEARCH$. First we subdivide X into a cubical lattice, discretizing the problem. We break the space up into cubes, or 'boxes', small enough that two points in a given cube are at most ϵ apart. The crucial fact about such a cube is that if an r' ball can be centered *somewhere* in the cube then an r ball can be centered *anywhere* in the cube. In particular, it might as well be centered at the center of the cube. We explore the unobstructed cubes by performing a depth-first search of the centers of the boxes[a]. To make this competitive, we restrict movement to an ellipsoid, and progressively increase the volume of the ellipsoid until a solution is found. Our virtual bounding ellipsoid is a direct generalization of the virtual ellipses of Gabriely and Rimon[7]. We will analyze these algorithms in the next section.

6.1. *Colors*

To begin, we introduce some terminology to make visualization of the algorithm's execution easier and to formalize some aspects of the algorithm. When our algorithms are being run, there are a few types of cubes that are encountered. We describe and associate a color to each type of cube:

- White: The initial condition of each cube. A cube is white if it is unexplored;
- Yellow: Bob's center can be at the center of the cube;
- Red: Too close to an obstacle: every point of the cube is within r' of an obstacle;
- Pink: the cube is completely outside of the virtual boundary.

As our algorithms run, they change White cubes into Yellow, Red, or Pink cubes, and (when increasing the size of the virtual boundary) Pink cubes back to White cubes.

[a]To be precise, a depth-first search of a dynamically generated spanning tree of the 1-skeleton of the dual of the cubical lattice.

6.2. *The Boxes Algorithm*

We are ready to define the Boxes algorithm and its companion algorithm, GraphTraverse. The Boxes algorithm solves modified $COVER$, NAV, and $SEARCH$. The GraphTraverse companion algorithm implements the depth-first search.

Boxes

- Define $l = \min\{\epsilon/2, \epsilon/\sqrt{n}\}$.
- Break X into a grid of axis-parallel boxes with side length l.
- Let C be the box currently containing Bob's center.
- Call GraphTraverse(C).
- If we are solving NAV or $SEARCH$
  - If T is in the current box
    - Travel in a straight line to T.
    - If an obstacle is encountered, there is no r'-path from S to T. Stop.
    - Otherwise, Bob has reached T. Stop.
  - Else
    - There is no r'-path from S to T.
- Else
  - Bob has successfully completed $COVER$.

GraphTraverse(C)

- If we are solving NAV or $SEARCH$ and T is in the current cube then return execution to Boxes.
- Let P be the current location.
- Move in a straight line toward the center of C.
- If we encounter an obstacle in the process
  - Color C Red.
  - Travel in a straight line back to P.
  - Return from this function call.
- Color C Yellow.
- Let *Adjacent* be the set of cubes sharing an $n-1$ dimensional face with C.
- While there are White cubes in *Adjacent*,
  - Pick a White neighbor, $D \in$ *Adjacent*.
  - GraphTraverse(D).

6.3. *The CBoxes Algorithm*

The CBoxes algorithm is an optimally competitive solution to the modified $SEARCH$ and NAV problems. It works by running Boxes in an ever expanding virtual boundary until that boundary is large enough that Boxes can find a path or determine that no path exists. This strategy is directly inherited from the CBUG algorithm of Gabriely and Rimon[7]. In the case of NAV, the virtual boundary is an ellipsoid with foci S and T, and in the case of $SEARCH$ it is a sphere centered at S. The shape of the virtual boundary, the rate at which it grows, and its initial size are actually not very important as far as the asymptotic competitiveness is concerned. The particular choices we have made make the analysis go smoothly.

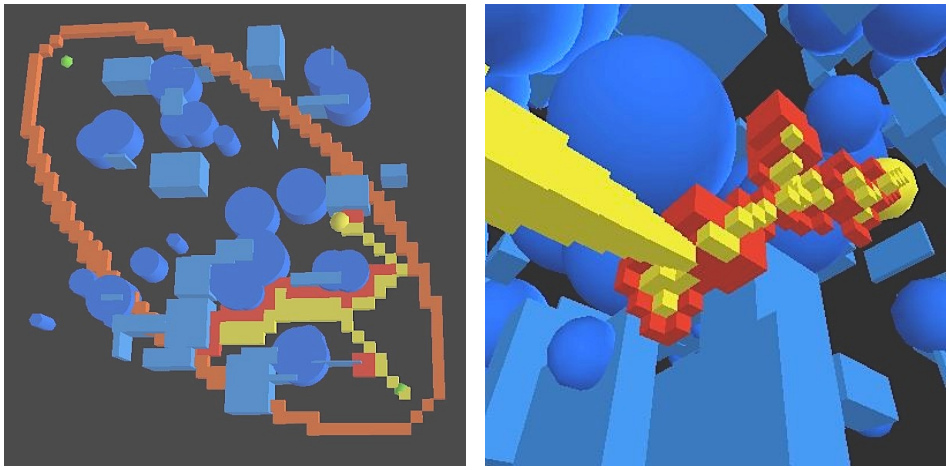
CBoxes

- If we're solving NAV
  - Define T' to be T.
- Else (we're solving $SEARCH$)
  - Define T' to be S.
- Set $a = d(S, T') + l$.
- While $1 = 1$
  - Let $\mathcal{E}$ be the solid ellipsoid defined by
    $$\mathcal{E} = \{p : d(S, p) + d(p, T') \leq a\}.$$
  - Color Pink all cubes that are completely outside of $\mathcal{E}$.
  - Call Boxes.
  - If Bob is in the same cube as T
    - If Bob is at T
      - Stop. We have completed the task.
    - Else
      - Stop. There is no r'-path from S to T.
  - Else if no neighbors of any Pink cubes were explored
    - Stop. There is no r'-path from S to T.
  - Color all cubes White.
  - Set $a = 2a$.

There is a simulation of this algorithm available at:

`http://www.math.binghamton.edu/sabalka/robotmotion`

See Figure 7 for some screenshots of that program.



(a) (b)

Fig. 7. Screenshots showing CBoxes being implemented. In Figure 7(a), CBoxes is being run on a 2-dimensional space, while Figure 7(b) is 3-dimensional. In both images, obstacles are the larger rectangles, spheres, and cylinders, Bob is the large lighter sphere, the start and target are smaller spheres, and the cubes show the state of Bob's memory. White cubes are not shown.

7. Correctness

In this section, we prove that our algorithms are correct. We will prove this rigorously, but essentially Boxes just performs depth first search on a graph, G_0, representing a discretization of the space. More specifically, the space X is broken into a grid of axis-parallel cubes with side length $l = \min\{\epsilon/2, \epsilon/\sqrt{n}\}$. Define G be the graph whose vertex set is the set of cubes, where there is an edge between two cubes if they share an $(n-1)$-dimensional face. Define G' to be the subgraph of G where two cubes are adjacent if there is an r-path between their centers. Then G_0 is the component of G' containing the start point. See Figure 8.

Lemma 2. *If a robot of radius $r' = r + \epsilon$ can be centered* somewhere *in a cube of side length $l = \min\{\epsilon/2, \epsilon/\sqrt{n}\}$ then a robot of radius r can be centered* anywhere *in that same cube.*

Proof. Since $l \leq \epsilon/\sqrt{n}$, the maximum distance between two points in a cube is at most ϵ. Thus a robot of radius centered within the cube is entirely contained in a radius r' robot centered anywhere else in the same cube □

Lemma 3. *If there is an r' path from S to a point in box B then Boxes visits the center of B when solving $COVER$.*

Proof. Call the r' path p. Let $C_1, C_2, \ldots, C_k$ be the sequence of cubes that p passes

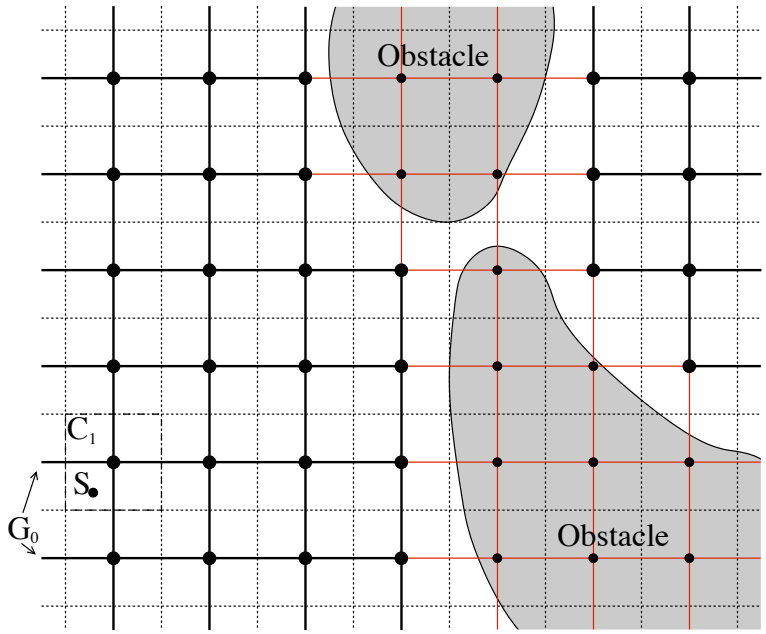


Fig. 8. Shown is a portion of some space X. The cubes of the grid are dotted, each with side length l. The graph G consists of the vertices and solid edges. The (r-neighborhoods of) obstacles are shown. Edges of G which are in the set $\mathcal{O}$ are thin. Thus, the graph G_0 is the connected component of thick solid edges which intersects the cube C_1.

through. Thus S is in cube C_1 and $C_k = B$; see Figure 9(a). Notice that C_i and C_{i+1} can be taken to share an $(n-1)$-dimensional face. If Boxes does not visit C_k, let j be the smallest index where C_j is not visited. Certainly $j > 1$. Then Boxes visited C_{j-1}. Since C_{j-1} and C_j share a face and both contain parts of the r' path p, we conclude from Lemma 2 that the path between their centers is an r-path. Thus, the last time Boxes visited C_{j-1}, Boxes must have found C_j to be colored something other than White, or it would have visited it. But Boxes only colors cubes Yellow or Red. Cube C_j must not have been Yellow since that would mean its center had been visited. Thus C_j was marked Red.

If a cube, D, is colored Red, this means that at some point Boxes tried to move from a cube C into D and in so doing encountered an obstacle. As we illustrate in Figure 9(b), in this circumstance, D cannot contain the center of a radius r' robot anywhere. To see this, let d be a point in D and let c be Bob's center upon hitting an obstacle point, o. Notice that the distance from c to o is r. The distance from c to the center of D is at most l, which is at most $\epsilon/2$. The distance from d to the

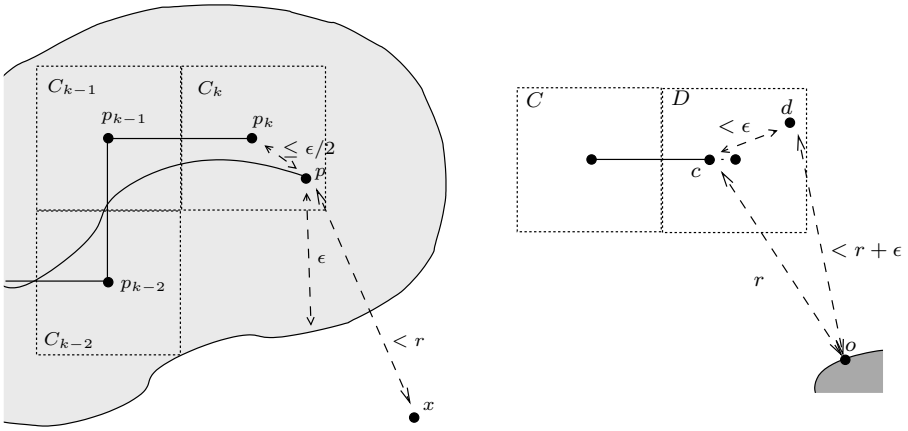


Fig. 9. In Figure 9(a), the point p is a point along some r'-path (shown) which comes within distance r of a point x. The shaded region is all points within ϵ of this r'-path, so Bob's center can be at any shaded point. If Bob occupies the point p_k, Bob will come close enough to x to explore it. In Figure 9(b), a simple triangle inequality verifies that if an obstacle is encountered in trying to move to the center of some cube D, then any point d of D is within $r + \epsilon$ of an obstacle.

center of D is at most $\epsilon/2$. By the triangle inequality, the distance from d to o is at most $r + \epsilon = r'$.

So since C_j is red, no part of any r' path can pass through it, contradicting the fact that C_j is on p. □

Theorem 8. *The Boxes algorithm solves the modified COVER problem.*

Proof.

Consider a point $x \in X$ that Bob's center is required by modified $COVER_n$ to come close to (within r'). Then there is an r'-path from S to a point y that has distance at most r from x. By Lemma 3, Boxes visits the center, c, of the box containing y. We have $d(c, x) \leq d(c, y) + d(y, x) \leq l/2 + r = \min\{\epsilon/2, \epsilon/\sqrt{n}\} + r < r + \epsilon = r'$. □

Theorem 9. *The Boxes algorithm solves the modified NAV and SEARCH problems.*

Proof. Boxes operates exactly the same for these algorithms as it does for $COVER$ except now if Bob finds himself in the Box containing T, he tries to move in a straight line to T. Thus, if there is an r'-path from S to T then by Lemma 3, Bob finds his way to the center of the box containing T. By Lemma 2, the straight line path to T is unobstructed, so Boxes makes it to T.

If there is no r'-path to T, Bob might nonetheless find an r-path, thus satisfying the requirements of NAV and $SEARCH$. Assume Bob does not find an r-path. Then since the space is bounded there are finitely many cubes that can be visited, so the algorithm eventually terminates. If Bob is not in the cube containing T, Boxes correctly concludes that that there is no r'-path from S to T. If Bob makes it to the cube containing T and encounters an obstacle while traveling to its center, we conclude from Lemma 2 that no r'-path passes through the cube and hence no r'-path ends at T. Thus Boxes correctly concludes that there is no r'-path from S to T. □

Theorem 10. *CBoxes solves NAV and $SEARCH$.*

Proof. First suppose that Bob eventually visits the cube containing T. Then Bob will try to move in a straight line to T. As we discuss in the proof of Theorem 9, if he succeeds then he has solved the task, and otherwise CBoxes correctly concludes that there is no r'-path from S to T.

On the other hand, if CBoxes never visits the cube containing T then because X is unbounded, the bounding ellipsoid will eventually be completely unreachable. After execution of Boxes with this very large bounding ellipsoid, Bob has not visited T or any of the boxes adjacent to pink boxes. Bob never saw a pink cube in the execution of this iteration of Boxes, so even if there were no bounding ellipsoid, Bob would have stopped without reaching T. Thus CBoxes is correct when it concludes that there is no r'-path from S to T. □

8. Competitiveness of Boxes and CBoxes

In this section we show that CBoxes is an optimally competitive solution to the modified NAV and $SEARCH$ problems. We also show that in a restricted set of spaces, Boxes is a competitive solution to $COVER$.

While Boxes solves the modified $COVER$ problem, it does not do so competitively in all circumstances. Informally, the problem is bottlenecks. For example, start in a very small room that has a single, narrow exit (ie a corridor of radius less than r') into a very big room (see Figure 10). Modified $COVER$ only requires coverage of the small room, so a robot that covers the large room is not optimal. Boxes may unwittingly make its way into the large room. We see two ways of alleviating this situation. The first is to provide Bob with a myopic visual sensor, able to detect bottlenecks: that is, able to detect all obstacle points within distance ϵ or so. To keep with the non-visual emphasis of this paper, we choose to analyze the second solution, by restricting our spaces to have no bottlenecks. We must therefore define precisely what it means to say that a space has a bottleneck.

Definition 9. We say that a space *has a bottleneck* if Bob can get within ϵ of some point via r-paths but not r'-paths. That is, if there are non-obstacle points within

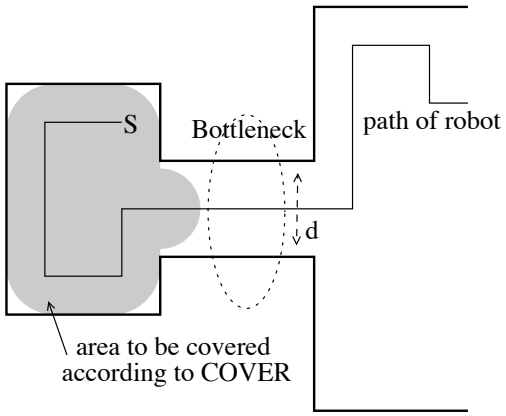


Fig. 10. In this figure, a bottleneck separates a small room from a large open space. Here, the width of the corridor d satisfies $2r < d < 2r + 2\epsilon$. The robot is only required to cover the shaded area, but Boxes might prescribe the path shown, traveling down the bottleneck.

distance r' of an r-path from S that are not within distance r' of any r'-path from S.

Theorem 11. *Assume $\epsilon < 2r$. For a given space X without bottlenecks, let l_{opt} be the length of an optimal path solving modified $COVER_n$. Then there exist constants $c(n, r, \epsilon)$ and $d(n, r, \epsilon)$ such that the length of the path generated by Boxes is at most $c(n, r, \epsilon)l_{opt} + d(n, r, \epsilon)$.*

Proof.

Boxes is basically a depth-first search of a subtree of the graph G, defined at the beginning of Section 7. The number of edges in a tree is the number of vertices minus 1. Just as in normal depth first search, Bob travels along each edge at most twice. The length of each edge is $l = \min\{\epsilon/2, \epsilon\sqrt{n}\}$. Let o be the optimal r'-path that solves $COVER$.

We claim that the number, c, of cubes that o passes through is at most $3^n(l_{opt}/l + 1)$. To see this, first consider a path of length l (the width of a box). The number of cubes that this path passes through is at most the maximum number of cubes that intersect an l-ball. Projecting this l-ball onto any dimension, we see that it intersects at most 3 cubes (in that dimension). Thus the ball is bounded by a bounding box, three cubes on a side, so a path of length l intersects at most 3^n cubes (in fact, the actual maximum is $3 \times 2^{n-1}$). Now, break o into several paths of length l and one of length at most l. If $l_{opt} = 0$, there is one such sub-path, and otherwise there are $\lceil l_{opt}/l \rceil$ sub-paths. In either case, the number of sub paths is at most $l_{opt}/l + 1$, and each one intersects at most 3^n cubes, proving the claim.

Now we claim that the number of cubes whose centers Boxes visits or tries to visit is $O(c)$. You should refer to Figure 11. Let C be such a cube. Then Bob's center

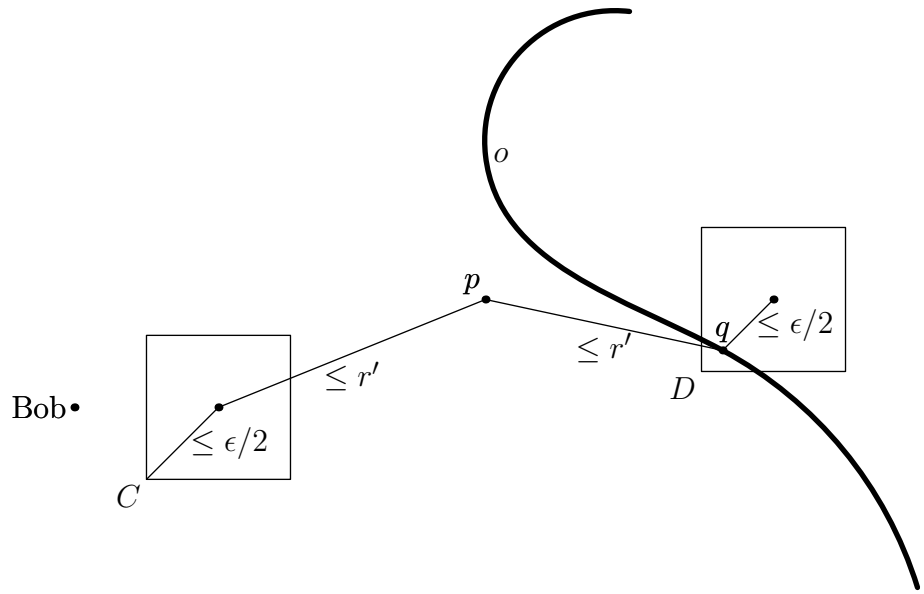


Fig. 11. We establish here that the union of the $(\epsilon/2 + r' + r' + \epsilon/2)$-neighborhood of the the centers of cubes visited by o (the optimal COVER path) contains all of the cubes visited by Boxes in spaces without bottlenecks.

comes within $l < \epsilon < r'$ of the center of C. Since there are no bottlenecks, C's center is within r' of a point p on an r'-path from S. Since o solves $COVER$, o contains a point q within r' of p. Let D be the cube containing q. By the triangle inequality, C is entirely contained in the radius $\epsilon/2 + r' + r' + \epsilon/2 \leq 3\epsilon + 2r$ ball centered at D. If $V(3\epsilon + 2r)$ is the volume of the n-sphere of radius $3\epsilon + 2r$, then there are at most $V(3\epsilon + 2r)/l^n$ cubes entirely contained in this sphere. This is certainly no more than $[2(3\epsilon + 2r)]^n/l^n$. Hence, the number of cubes visited successfully or unsuccessfully by Bob is at most $c[2(3\epsilon + 2r)/l]^n$.

The length of Bob's path is at most $2l$ times the number of Boxes Bob visits or tries to visit. This is at most $2lc[2(3\epsilon + 2r)/l]^n \leq 2l3^n(l_{opt}/l + 1)[2(3\epsilon + 2r)/l]^n = (2 \cdot 3^n[2(3\epsilon + 2r)/l]^n)l_{opt} + (2^{n+1}3^n(3\epsilon + 2r)^n/l^{n-1})$. □

Corollary 2. *When restricted to spaces without bottlenecks, Boxes is an optimally competitive solution to modified $COVER$.*

We now compute an upper bound on complexity for CBoxes.

Theorem 12. *Let l_{opt} be the length of the optimal r'-path from S to T. Then the length of the path generated by CBoxes is at most*

$$cl_{opt}^n + d,$$

where c and d are functions of n and ϵ.

Proof. We will first handle NAV. Reasoning as we did in the proof of Theorem 11, we see that during any iteration of Boxes, the path generated by CBoxes is at most $2l$ times the number of cubes not entirely outside of the ellipsoid $\mathcal{E}$. For an easy and succinct upper bound on the number of such cubes, we note that $\mathcal{E}$ is contained in the axis-parallel hypercube with side length $2a$ centered at the midpoint of S and T. This hypercube intersects at most $(2a/l+2)^n$ cubes in the partition of X. Since a is initally chosen to be $d(S,T)+l$, we have $a \geq l$ and hence $a/l \geq 1$. Thus the hypercube intersects at most $(2a/l+2)^n \leq (2a/l+2a/l)^n = (4a/l)^n$ cubes of the partition of X.

Define a_0 to be the initial value of a. Suppose we start at iteration 0 and that CBoxes finds T during iteration i. Then $a = 2^i a_0$. If $i > 0$, then a is finally large enough that GraphTraverse finds its way to T, while the previous bounding ellipsoid is not large enough. Note that the ellipsoid with parameter a is precisely the locus of points along paths from S to T of length a. In particular, $l_{opt} > a/2 = 2^{i-1}a_0$, since otherwise the optimal path would be entirely within the previous bounding ellipsoid, and hence Boxes would have found a path. Thus the total number of cubes that CBoxes visits is at most

$$\begin{aligned}
\sum_{j=0}^{i} \left(\frac{4 \cdot 2^j a_0}{l}\right)^n &= \left(\frac{4a_0}{l}\right)^n \sum_{j=0}^{i} 2^{jn} \\
&= \left(\frac{4a_0}{l}\right)^n \frac{(2^{i+1})^n - 1}{2^n - 1} \\
&< \left(\frac{4}{l}\right)^n \frac{(2^{i+1}a_0)^n}{2^n - 1} \\
&< \frac{1}{2^n - 1}\left(\frac{16 l_{opt}}{l}\right)^n.
\end{aligned}$$

If $i = 0$ then one of two things can happen: either $d(S,T) > l$ or $d(S,T) \leq l$. If $d(S,T) > l$ then $a = d(S,T) + l < 2d(S,T) \leq 2l_{opt}$. Thus $l_{opt} > a/2$ and the analysis above applies. If $d(S,T) \leq l$ then $a = d(S,T) + l \leq 2l$. In this case, the total number of boxes visited by CBoxes is at most $(2a/l+2)^n \leq 6^n$. In any case, the total number of boxes visited is at most

$$\frac{1}{2^n - 1}\left(\frac{16 l_{opt}}{l}\right)^n + 6^n.$$

We multiply by $2l$ to get the a bound on the total distance spent travelling between centers of cubes. Since we move to and from the centers of the first and last cubes, we may need to travel an additional length at most ϵ. And thus the total distance traveled is no more than

$$2l\left(\frac{1}{2^n - 1}\left(\frac{16 l_{opt}}{l}\right)^n + 6^n\right) + \epsilon.$$

Since l is a function of ϵ and n, our claim is established for NAV.

A similar analysis works for $SEARCH$. The only significant difference is that we explore the sphere of radius $a/2$ centered at S rather than an ellipsoid with foci S and T. Notice that this sphere contains all of the length $a/2$ paths starting at S. Thus if T is found on iteration $i > 0$, it was not found on iteration $i - 1$ and we conclude that $l_{opt} > (2^{i-1}a_0)/2 = 2^{i-2}a_0$. This is twice as bad as in the NAV case and it results in a leading constant that's twice as bad. Nonetheless, we reach the desired conclusion. □

Corollary 3. *CBoxes is optimally competitive when solving both the modified $SEARCH$ and modified NAV problems.*

9. Observations and Improvements

We are able to make a number of improvements to the algorithms described.

9.1. *Sampling Improvement for COVER, SEARCH, and NAV*

For this paper, we have chosen to discretely sample the unknown environment X via the centers of a grid of cubes. These centers form a lattice (in fact, a cubical lattice: they are the vertices of the dual cubical tiling). Let the *diameter* of a lattice denote the maximum distance between points in a primitive cell of the lattice – that is, a fundamental domain of the quotient of $\mathbb{R}^n$ by the translational symmetries of the lattice. The only mathematical properties of the lattice we used were that every point in $\mathbb{R}^n$ was within r of a point of the lattice, and that the diameter was at most ϵ. In fact, other lattices would work. One should be able to choose a more efficient lattice structure to sample the space with fewer lattice points. This problem is closely related to that of sphere-packing. Duals of lattices associated to optimal sphere-packing seem to reduce the number of lattice points per volume needed. In particular, using the duals of the lattices associated with Gauss's hexagonal sphere-packing in dimension 2 or close packings in dimension 3 should yield better results in those dimensions. Indeed, if one could find a good way of encoding it, even a good irregular sphere-packing would yield a better sampling of X.

9.2. *Taking Diagonals Improvement for COVER, SEARCH and NAV*

Our complexity estimates in part relied on the distances between centers of cubes sharing a codimension-1 face. However, one can obtain similar estimates even if one allows Bob to travel from the center of one cube to the center of any other adjacent cube, sharing a face of arbitrary dimension. This may worsen the complexity estimates, but should improve average-case runtime by a factor of up to $\sqrt{n}$.

9.3. *Noticing T Improvement for SEARCH*

While trying to solve the $SEARCH_n$ problem, it will occasionally happen that Bob finds out where T is but cannot move its center directly to T because of nearby obstacles (for instance, when $\epsilon < (\sqrt{2}-1)r$ and T is close to the center of a gap in obstacles slightly smaller than Bob). Whenever T is discovered, the CBoxes algorithm should begin to treat $SEARCH_n$ as if it were a NAV_n problem, and use the improvements below for choosing the cube D referenced in the GraphTraverse algorithm and travelling expediently to T.

9.4. *Maximal Coloring Improvement for COVER, SEARCH and NAV*

One straightforward improvement to the algorithms is to take full advantage of knowing a point on the boundary of an obstacle. Currently, if Bob runs into an obstacle, only the cube D that Bob was trying to get to is colored Red. But Bob knows that many other cubes should also be colored Red. The Maximal Coloring Improvement is, whenever an obstacle point is encountered, to color all cubes Red that have all corner points within distance r' of the given obstacle point. As a cube is convex, this is equivalent to saying a robot of radius r' with center in the cube will intersect the obstacle point.

If we are solving $SEARCH$ or NAV and we know T is in a Red cube, stop. T cannot be reached.

This improvement will cause many White and Pink cubes to be colored Red, and occasionally will cause a Yellow cube, C_Y, to be colored Red. Bob can take this into account by checking to see if C_Y is on the path from the cube containing S (call it C_S) to the current cube C in the spanning tree generated by CBoxes. If it is, then Bob can immediately return to the cube before C_Y, ignoring any White neighbors of cubes between C_Y and C. Either these neighbors will be explored via some other route, or they are not reachable by a robot of radius r' and so need not be explored.

9.5. *Gray Improvement for NAV*

For the modified NAV_n problem, another improvement may be made by adding a new color. As written, GraphTraverse will explore every possible White cube, even if exploration would give Bob no new information on how to get to T. For instance, consider a space with a very large sphere about S as an obstacle separating S from T. Place a hole in the sphere so that a robot of radius $r+\epsilon$ can fit through. Imagine that Bob has explored the entire inner boundary of the sphere, and finally reaches the hole. Clearly, Bob should exit the sphere, as exploring any more boxes inside the sphere would just require backtracking, and Bob knows it. This knowledge can be incorporated into the algorithm as follows. We create a new color designation:

- Gray: Never to be explored.

Bob doesn't know what's in a Gray cube, but Bob will never go into one. If there is ever a connected component Z of the union of all White cubes that doesn't contain T, color every cube in Z Gray. Any path through centers of cubes to T through Z can be replaced by a path not through Z (eventually, entirely through Yellow cubes).

When combined with the Pink color designation for the CBoxes algorithm, note that which cubes are White and which are Gray should be recomputed by CBoxes between executions of Boxes.

9.6. *Greedy Improvement for NAV*

Coloring cubes Gray can potentially save Bob unnecessary exploration time by helping decide which cube D to explore next while executing the GraphTraverse algorithm. In fact, there is an even more efficient way of choosing which cube D to explore in the GraphTraverse algorithm. At every step, choose D as follows. If there is a path from the current cube C through centers of cubes to T such that all cubes on the path are colored White except possibly at the endpoints, then find a shortest such path γ. Choose D to be the next cube along γ from C. If there is no path from C though centers of White cubes to T, there is no need to explore any adjacent unexplored cubes. In fact, with the Gray Improvement, there can be no adjacent White cubes: all adjacent unexplored cubes can be colored Gray. In this situation, GraphTraverse will have Bob back up to the last cube which is not surrounded by non-White cubes. If ever there does not exist a path from a previously explored Yellow cube through White cubes to T, stop: no path exists from S to T for a robot of radius $r + \epsilon$. In other words, choose D greedily, and this is guaranteed to work. Note this way of choosing D does not actually require the introduction of the color Gray, and this improvement supersedes the Gray Improvement.

We note that this improvement can in particular be applied to CBoxes in 2-dimensional environments. Compared to CBUG, CBoxes has two drawbacks: the requirement of nonconstant memory, and the introduction of the clearance parameter ϵ, particularly in the dependence on ϵ in the upper bound on competitiveness. However, both algorithms are optimally competitive with respect to modified l_{opt}, and in many environments the Greedy Improvement will help CBoxes by always proceeding towards the target instead of exploring the entirety of an obstacle.

9.7. *Wide Open Spaces Improvement for COVER, SEARCH and NAV*

Currently, our algorithms use very small cubes to explore X. If X has a large open area to explore, this can be wasteful. Just as the Maximal Coloring Improvement takes advantage of where obstacles are, we should also take advantage of where obstacles are not. We can do this with the following observation. Let N denote the r'-neighborhood of the center of a White cube C. If N is contained in the union of r-neighborhoods of all (nearby) centers of Yellow cubes, then we know even without

visiting C that C should be colored Yellow (or some color designating that the cube need not be visited).

If the cubes which are visited are chosen carefully, this can greatly reduce the number of cubes which need to be explored. We note, however, that this improvement is mostly unnecessary when using the following Subdivision Improvement, which is similar in essence.

9.8. *Subdivision Improvement for COVER, SEARCH and NAV*

Our algorithm as stated subdivides the ambient space into cubes which are as small as necessary to prove our theorems. But boxes of side length less than $\epsilon/\sqrt{n}$ can be too small in large, sparsely obstructed environments. The only time it was necessary for us to use such small cubes was when proving our algorithm successfully executes in task instances which require Bob to pass through tight corridors, of diameter greater than $r + \epsilon$ but not by much. We may search for such tight spaces using a much coarser exploration grid (i.e. much larger boxes), and only subdivide one of these larger boxes into smaller boxes when necessary.

We present here the precise subdivision algorithm for NAV. The other algorithms are similar.

Begin by breaking X into a grid of axis-parallel cubes of side length l' on the order of r, so that the centers of adjacent cubes (sharing a face with arbitrary codimension) are no more than $2r$ apart – say, $l' := r/(2\sqrt{n})$. Start with a fixed-radius bounding ellipsoid and execute CBoxes with the following changes. When an obstacle point, p, is encountered, color cubes Red like in the Maximal Coloring Improvement. Subdivide a non-Red cube into 3^n subcubes if its side length is greater than l and it intersects the r' ball centered at p. Color each of the newly created cubes as appropriate: White by default, Red if all corners are within r' of the obstacle point, Pink if entirely outside of the virtual bounding ellipsoid, and Yellow when the center has previously been visited. Subdivided cubes are adjacent to any cube with which they share any portion of a face. If execution stops without reaching T, color every Yellow cube White and restart with the new set of cubes. Continue until no new subdivisions are created. If at this point T has not been reached, expand the bounding ellipsoid and repeat.

Proof of correctness. Suppose the ellipsoid is large enough to contain an r' path, ρ, from S to T. Then we claim that the algorithm finds T before expanding the ellipsoid again. Suppose not. Then the algorithm goes through an iteration without finding T and without subdividing a cube. Since ρ is an r'-path, it never enters a Red cube. Thus ρ either never leaves Yellow cubes or it enters a White cube, W, adjacent to a Yellow cube, Y.

In the first case, Bob visited the center of the cube C containing T but was unable to move straight from the center to T, so Bob must have hit an obstacle point, p. The r' neighborhood of p contains Bob's center and thus intersects C.

Since ρ also intersects C, C is not entirely contained in the r' neighborhood of p. Thus C should have been subdivided, and we have reached a contradiction.

In the second case, Bob attempted to move from the center of Y to the center of W but encountered an obstacle point. Similar to the reasoning in the previous paragraph, either Y or W should have been subdivided. □

The Subdivision Improvement means that, for the vast majority of the time, our algorithms will quickly move about in large steps. Although we give no analysis here, the length of the path travelled using the Subdivision Improvement can be made to be at most a constant times the length of the path travelled without the improvement by limiting the number of times the algorithm can restart within each ellipsoid. Thus, the upper bound on competitiveness with the Subdivision Improvement is in the same complexity class as the basic algorithm. In typical cases, with the Subdivision Improvement our algorithms should not only run faster but need far less memory to execute.

References

1. Ricardo A. Baeza-Yates, Joseph C. Culberson, and Gregory J. E. Rawlins. Searching in the plane. *Inform. and Comput.*, 106(2):234–252, 1993.
2. Piotr Berman. On-line searching and navigation. In *Online algorithms (Schloss Dagstuhl, 1996)*, volume 1442 of *Lecture Notes in Comput. Sci.*, pages 232–241. Springer, Berlin, 1998.
3. Avrim Blum, Prabhakar Raghavan, and Baruch Schieber. Navigating in unfamiliar geometric terrain. *SIAM J. Comput.*, 26(1):110–137, February 1997.
4. David G. Caraballo. Areas of level sets of distance functions induced by asymmetric norms. *Pacific J. Math.*, 218(1):37–52, 2005.
5. Joseph Carsten, Arturo Rankin, David Ferguson, and Anthony Stentz. Global path planning on-board the mars exploration rovers. In *IEEE Aerospace Conference*, 2007.
6. Amos Fiat and Gerhard J. Woeginger, editors. *Online algorithms: the state of the art*, volume 1442 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1998. Papers from the Workshop on the Competitive Analysis of On-line Algorithms held in Schloss Dagstuhl, June 1996.
7. Yoav Gabriely and Elon Rimon. Cbug: A quadratically competitive mobile robot navigation algorithm. *IEEE Trans. on Robotics and Automation*, 6:1451–1457, 2008.
8. C. Icking, T. Kamphans, R. Klein, and E. Langetepe. On the competitive complexity of navigation tasks. In *Sensor Based Intelligent Robots, Lecture Notes in Computer Science*, volume 2238, pages 245–258. Springer Verlag, 2002.
9. C. Icking and R. Klein. Competitive strategies for autonomous systems. In H. Bunke, T. Kanade, and H. Noltemeier, editors, *Modelling and Planning for Sensor Based Intelligent Robot Systems*, pages 23–40. World Scientific, Singapore, 1995.
10. L.E. Kavraki, M.N. Kolountzakis, and J.-C. Latombe. Analysis of probabilistic roadmaps for path planning. *IEEE Transactions on Robotics and Automation*, 14(1):166–171, Feb. 1998.
11. J. C. Latombe. *Robot Motion Planning*. Kluwer Academic, Boston, MA, 1991.
12. Steve Lavalle. *Planning Algorithms*. Cambridge University Press, 2006.
13. Vladimir J. Lumelsky and Alexander A. Stepanov. Path-planning strategies for a point

mobile automaton moving amidst unknown obstacles of arbitrary shape. *Algorithmica*, 2(4):403–430, 1987. Special issue on robotics.

14. C. H. Papadimitriou and M. Yannakakis. Shortest paths without a map. *Theoretical Computer Science*, 84:127–150, 1991.
15. D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Comm. of the ACM*, 28(2):202–208, 1985.