Illinois Wesleyan University
Digital Commons @ IWU

John Wesley Powell Student Research Conference

2010, 21st Annual JWP Conference

Apr 10th, 10:00 AM - 11:00 AM

Optimization and Analysis of a Robotic Navigational Algorithm

Derek Carlson
Illinois Wesleyan University

Joshua Brown Kramer, Faculty Advisor
Illinois Wesleyan University

Follow this and additional works at: https://digitalcommons.iwu.edu/jwprc

Part of the Artificial Intelligence and Robotics Commons

Carlson, Derek and Brown Kramer, Faculty Advisor, Joshua, "Optimization and Analysis of a Robotic Navigational Algorithm" (2010). *John Wesley Powell Student Research Conference*. 5.
https://digitalcommons.iwu.edu/jwprc/2010/oralpres2/5

This Event is protected by copyright and/or related rights. It has been brought to you by Digital Commons @ IWU with permission from the rights-holder(s). You are free to use this material in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you need to obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/ or on the work itself. This material has been accepted for inclusion by faculty at Illinois Wesleyan University. For more information, please contact digitalcommons@iwu.edu.
©Copyright is owned by the author of this document.

Enhancement of a Simulated Robotic Navigation Algorithm

Derek Carlson

Illinois Wesleyan University

Abstract

The problem of robot navigation involves planning a path to move a robot from a start point to a known target point within an obstacle course. The efficiency of such an algorithm can be measured in several ways. For instance, Lumelsky and Stepanov measure the length of the path taken in terms of obstacle perimeters. Gabriely and Rimon compare their two-dimensional algorithm's efficiency to that of the optimal algorithm. Brown Kramer and Sabalka expand upon the work of Gabriely and Rimon to produce an algorithm for dimensions greater than two. The primary objective of this research was to improve Brown Kramer and Sabalka's algorithm called Boxes, which performs a depth-first search of a discretized obstacle space. We implemented an improvement called expansion of the ellipsoid, enabling the user to run an instance of Boxes' counterpart, CBoxes. We also implemented enhancements known as subdivision of the obstacle space and the maximal coloring improvement. Furthermore, the user is given the option of not running an improvement called the greedy heuristic, and we measure the change in performance caused by all of these improvements. Our data indicates that the greedy heuristic drastically improves algorithm performance in regards to path length and that subdivision and maximal coloring also improve performance, but to a lesser degree.

Contents

1 Introduction **1**

2 Background **1**
2.1 Bug1 and Bug2 . . . 2
2.1.1 Bug1 . . . 2
2.1.2 Analysis of Bug1 . . . 4
2.1.3 Bug2 . . . 5
2.1.4 Analysis of Bug2 . . . 5
2.2 CBug . . . 7
2.2.1 Analysis of CBug . . . 9
2.2.2 Competitiveness . . . 9
2.3 Boxes and CBoxes . . . 9
2.3.1 Boxes . . . 10
2.3.2 GraphTraverse . . . 11
2.3.3 Analysis of Boxes . . . 12
2.3.4 CBoxes . . . 12
2.3.5 Analysis of CBoxes . . . 14

3 Improvements **14**
3.1 User Interface . . . 14
3.2 Taking Diagonals . . . 14
3.3 Greedy Heuristic . . . 14
3.4 Expansion of the Ellipsoid . . . 15
3.5 Maximal Coloring . . . 15
3.6 Subdivision . . . 16

4 Performance **16**
4.1 Experiments . . . 16
4.2 Greedy Heuristic Data and Analysis . . . 17
4.3 Maximal Coloring Data and Analysis . . . 17
4.4 Subdivision Data and Analysis . . . 18

5 Conclusions **19**

6 Future Work **19**

7 Acknowledgements **21**

8 Bibliography **22**

Appendices **23**

A Python Code **23**

List of Figures

1 The red line displays the robot's initial path using the Bug1 algorithm. The blue line represents the path after circling the obstacle. Note that the robot takes the upper, shorter path. H and L indicate hit and leave points respectively. 3
2 A course in which Bug1 will perform poorly. The blue lines indicate the path taken by an automaton using Bug1. 4
3 An illustration of the Bug2 algorithm. The red line is the line from start to target, while the blue line is the path taken by the robot. 6
4 A course in which Bug2 can perform poorly. The red line represents the initial path. The blue line represents the path taken after the second contact with the obstacle. n_i for this example is 6. 6
5 CBug improves upon Bug1. The red line indicates the path the robot will take. 8
6 A screenshot of Boxes running in three dimensions. 12
7 A screenshot of CBoxes running in 2D 13

List of Tables

1 Path lengths generated with and without Greedy improvement . 17
2 Path lengths generated with and without maximal coloring . . . 17
3 Path lengths generated with and without subdivision and without maximal coloring 18

1 Introduction

Lumelsky and Stepanov [1] present an overview of the problem of robot navigation. Robot navigation is a path planning problem that can be considered in two ways; Possession of complete information or incomplete information. Possession of complete information means that the obstacles' shapes, sizes, and locations are all known when a path is computed. When complete information is possessed, the problem is sometimes called the piano movers problem [4] and path planning is more of a problem of efficiency. A calculation occurs once in an offline state, meaning that all of the data regarding the obstacle space is used to create a path before the robot executes that path. Thus, the complexity of the problem lies in computing a solution efficiently and the measure of performance of an algorithm is the time taken to calculate a path. However, when incomplete information is possessed, the focus shifts from runtime to path length. Instead of making a one-time offline computation, the automaton has to constantly gather data and calculate the path on the fly. This is known as an online algorithm, which is the type of algorithm we focus on.

In this thesis, we study several online robotic navigation algorithms. We focus on the measure of each algorithm's performance by examining the upper and lower bounds on path length. In particular, we examine the Bug1, Bug2, CBug, Boxes, and CBoxes algorithms and their efficiency. We first review the highly similar Bug1 and Bug2 algorithms of Lumelsky and Stepanov [1] in Section 2.1. We then turn our attention to the CBug algorithm of Gabriely and Rimon [2] in Section 2.2. Primarily, we center our examination on the Boxes and CBoxes algorithms of Brown Kramer and Sabalka [3], which, unlike the other algorithms, are capable of handling dimensions greater than two. We review CBoxes and Boxes in Section 2.3.

One goal of this research was to improve an existing implementation of the Boxes algorithm. One such enhancement was the implementation of an improvement called expansion of the ellipsoid, allowing the user to run the CBoxes algorithm. The main goal was to improve the performance of the algorithm by implementing improvements. In particular, we wanted to examine what are known as the greedy heuristic, maximal coloring, and subdivision improvements. Measuring the boost in performance for each improvement was another aim of our research. The improvements are presented in Section 3. We then present and analyze data on algorithm performance with the implemented improvements in Section 4. Lastly, our conclusions are presented in Section 5.

2 Background

In order to fully understand and accurately modify the Boxes algorithm of Dr. Joshua Brown Kramer and Dr. Lucas Sabalka, study of other robotic navigational algorithms was necessary. Brown Kramer and Sabalka expanded upon the work of Yoav Gabriely and Elon Rimon, who created the CBug algorithm. CBug will be examined later as we will first study the algorithm CBug was

based on. We begin our review by examining the Bug1 and Bug2 algorithms of Vladimir Lumelsky and Alexander Stepanov.

2.1 Bug1 and Bug2

Vladimir J. Lumelsky and Alexander A. Stepanov [1] presented two online algorithms for robotic navigation: Bug1 and Bug2. Both algorithms use a constant amount of memory and the only difference between the two is how each handles traversal of obstacles. The Bug algorithms serve as the foundation for the other algorithms and methodologies we will later examine.

The approach of Lumelsky and Stepanov is to assume the robot is a point in a two-dimensional space. The automaton has no knowledge of the obstacle space, meaning it does not know the locations, size, or shape of the obstacles present in the space. The only information known to the robot is the location of the target point and the current location of the robot. The task is to navigate the automaton from its current location to the target point.

A number of terms need to be defined in order to understand and analyze how the Bug algorithms work, which will be described here. D is the distance from the start point to the target point. P is the total length of the path taken by the robot to reach the target point and p_i is the perimeter of the i-th obstacle contacted by the robot. When the robot makes contact with the i-th obstacle, a hit point, H_i, is defined. Similarly, a leave point, L_i, is defined when the robot leaves an obstacle. Lastly, S denotes the start point and T denotes the target point.

2.1.1 Bug1

With those definitions in place, we will now describe Bug1. From the start or any leave point, the robot moves in a straight line toward the target. If the target is reached, then the procedure is completed. Otherwise, the robot encounters an obstacle, defining a hit point H_i. The robot then moves around the obstacle, essentially tracing it. While it is traversing the obstacle boundary, the robot is calculating its distance to the target at every point, and continuously updating the location of the shortest distance to the target in a register. The point closest to the target on the obstacle boundary is called Q_m. Once the robot has completely traced the obstacle (i.e. it returns to H_i), it defines the leave point L_i, which is equivalent to Q_m. If the robot crosses an obstacle at a point L_i, meaning it contacts the same obstacle it has just traversed, then the target is unreachable. For example, one could envision the target located at the center of a circular obstacle. Upon completion of tracing the circle obstacle and moving to Q_m, the robot would bump into the same obstacle. This is a sufficient test due to the fact that obstacles are not allowed to touch one another, so encountering an obstacle at the leave point indicates that either the robot or target point is trapped by an obstacle. Thus, if the target is unreachable, it stops. Otherwise, it traces around the obstacle again, taking the shorter path around the obstacle to Q_m/L_i (the path lengths are stored in registers as

well). At this point, the procedure repeats until the target is reached or deemed unreachable. Pseudocode for the algorithm is presented below and a Figure 1 illustrates how the Bug1 algorithm works.

Bug1(S, T)

$i = 1$
$L_0 = S$
C = current position of the robot
While $C \neq T$
    Move toward T in a straight line
    Update C
    If T is reached
        End execution
    If an obstacle is encountered
        H_i = hit point
        $Q_m = H_i$
        Do while $C \neq H_i$
            Move along the obstacle boundary
            Update C
            If $C = T$, end execution
            If C's distance to $T < Q_m$'s distance to T
                $Q_m = C$
        $L_i = Q_m$
        If test for target reachability fails
            End execution
        Else
            Take the shorter path to L_i
            i++

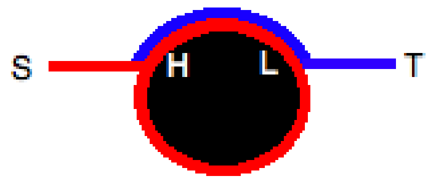


Figure 1: The red line displays the robot's initial path using the Bug1 algorithm. The blue line represents the path after circling the obstacle. Note that the robot takes the upper, shorter path. H and L indicate hit and leave points respectively.

2.1.2 Analysis of Bug1

Lumelsky and Stepanov [1] show Bug1 has an upper bound on the path length of $D + 1.5 \times \sum_i p_i$, meaning that the length of the path generated by the algorithm is the sum of all the obstacle perimeters times 1.5 plus the distance from the starting point to the target point. This is because the robot goes all the way around an obstacle once, then returns to the point Q_m. Moving to Q_m will take at most half the perimeter of the obstacle, since the robot will always choose the shorter path. D is added to account for the straight line segments in between obstacles. Since any leave point is closer to the target than the hit point (otherwise the target is unreachable), D is the maximum length that is traveled in straight line segments by the robot.

There are certain cases where Bug1 can perform poorly. For instance, consider the case where the automaton starts very close to the target, but with a very lengthy obstacle in between them as in Figure 2. The robot will move toward T, but contact the obstacle, causing it to traverse at most 1.5× the obstacle perimeter. The robot started close to the target, yet took an exceedingly long path in order to reach it. Stepanov and Lumelsky address this issue with another algorithm, Bug2.

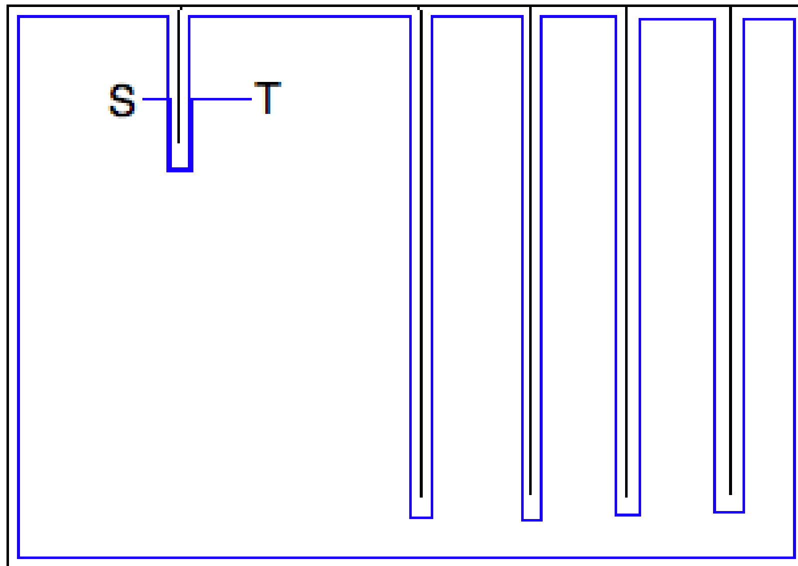


Figure 2: A course in which Bug1 will perform poorly. The blue lines indicate the path taken by an automaton using Bug1.

2.1.3 Bug2

Now for a description of Bug2. From the start or a leave point, the robot will move towards T in a straight line. If the target is reached, it is finished. Otherwise, the robot encounters an obstacle. Thus, a hit point, H_j, is defined, where j indicates the j-th contact with an obstacle. Like Bug1, the robot traces the obstacle. If the target is reached along the obstacle boundary, the procedure stops. Otherwise, the robot will trace the obstacle until it encounters a point such that the line from S to T intersects that point. If that point is closer to the target than the hit point, H_j, and it does not cross the obstacle, then a leave point, L_j is defined and the process starts over. If the robot does not reach such a point, i.e. reaches H_j, then the target is unreachable and the procedure stops. Pseudocode for Bug2 is provided below and an illustration of Bug2 is provided in Figure 3.

```
                              Bug2(S, T)

j = 1
L_0 = S
C = current position of the robot
While C ≠ T
    Move toward T and update C
    If T is reached
        End execution
    If an obstacle is encountered
        H_j = hit point
        Move along obstacle boundary
        Do while C ≠ H_j
            If C = T, end execution
            If C is on the line from the S to T
                If the distance from C to T < distance from H_j to T
                    If the robot does not cross an obstacle at C
                        L_j = C
                        j++
                        Break out of do loop
```

2.1.4 Analysis of Bug2

Lumelsky and Stepanov [1] prove the Bug2 algorithm has an upper bound on path length of $D + \sum_i \frac{n_i p_i}{2}$, where n_i is the number of times that the straight-line from S to T crosses the i-th obstacle. As mentioned in the analysis of Bug1, D accounts for the straight-line segments traveled by the automaton in between traversal of obstacles. As for the obstacle perimeter summation, the robot will pass a point of an obstacle at most $n_i/2$ times, as cycles are generated

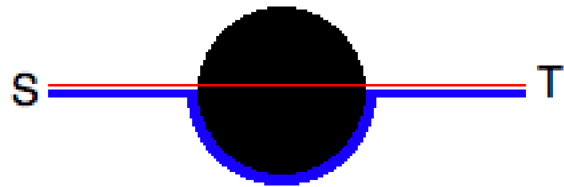


Figure 3: An illustration of the Bug2 algorithm. The red line is the line from start to target, while the blue line is the path taken by the robot.

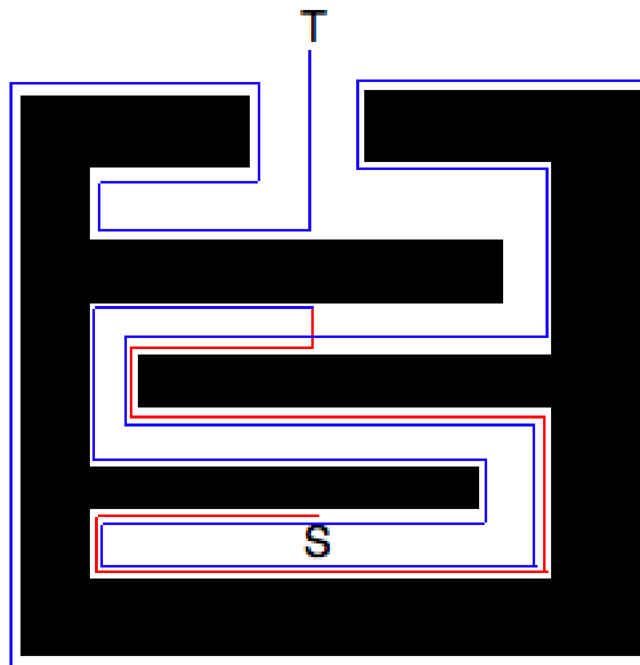


Figure 4: A course in which Bug2 can perform poorly. The red line represents the initial path. The blue line represents the path taken after the second contact with the obstacle. n_i for this example is 6.

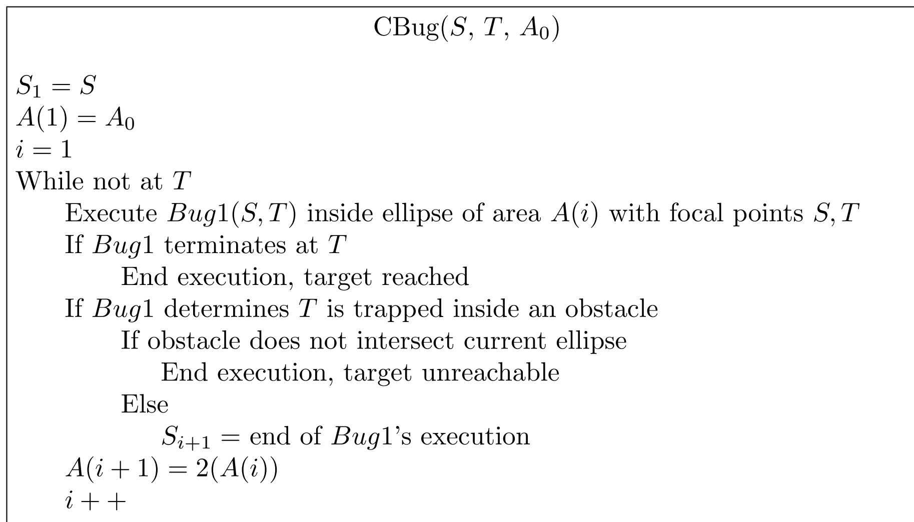
by hit and leave point pairs and n_i represents the maximum number of hit and leave points. For most simple obstacles, n_i will be two, but for more complex obstacles, n_i can be much greater, an example of which is provided in Figure 4. Bug2 can have cycles or repeated segments, which can only be created by hit and leave point pairs. Therefore, the max number of times a segment can be traveled again is the number of hit and leave point pairs, which is equal to $n_i/2$. By multiplying $n_i/2$ by p_i, the maximum possible distance traveled while traversing the i-th obstacle is obtained. By summing this for all obstacles and adding D, the upper bound on path length is obtained.

Because Bug2 leaves an obstacle perimeter when it is on the straight-line from S to T, the problem previously described for Bug1 is nullified. However, now Bug2 can have cycles, repeating sections of the obstacle boundary several times. An example of an obstacle that can produce cycles is shown in Figure 4. So while Bug1 and Bug2 are useful algorithms, they still have notable flaws. Furthermore, the performance analysis wasn't as helpful as it could be because the algorithms were evaluated with regard to the obstacle perimeters, which is not a reliable measure of the difficulty of an obstacle course. Enter Gabriely and Rimon [2], who introduce CBug and competitiveness.

2.2 CBug

The first contrast between the aforementioned Bug algorithms and CBug is that CBug has a circular robot of diameter D, rather than a point. The most striking difference is that CBug uses an ellipse to travel in an environment. This ellipse has focal points at S and T and the area of the initial ellipse is A_0. The boundary of the ellipse is treated as a virtual obstacle. That is, the robot is to treat the boundary as an obstacle, but the robot also knows when it makes contact with the ellipse that it is not a physical obstacle. There are other differences, such as how the algorithm's performance is analyzed, but those will be discussed later. First, we will explain the CBug algorithm.

The algorithm begins by creating the previously described ellipse. This is followed by the application of the Bug1 algorithm in the initial ellipse. If the target is reached, then the process stops. Otherwise, it is determined that the automaton is trapped, either by obstacles or the ellipse. Essentially, if the automaton did not touch the bounding ellipse, the target is unreachable. If it is deemed that the target is actually reachable, but not in the current ellipse, then the ellipse is expanded by doubling its area. The process repeats from this point on. Pseudocode for CBug is provided on the next page.



CBug(S, T, A_0)

$S_1 = S$
$A(1) = A_0$
$i = 1$
While not at T
    Execute $Bug1(S, T)$ inside ellipse of area $A(i)$ with focal points S, T
    If $Bug1$ terminates at T
        End execution, target reached
    If $Bug1$ determines T is trapped inside an obstacle
        If obstacle does not intersect current ellipse
            End execution, target unreachable
        Else
            S_{i+1} = end of $Bug1$'s execution
    $A(i+1) = 2(A(i))$
    $i++$

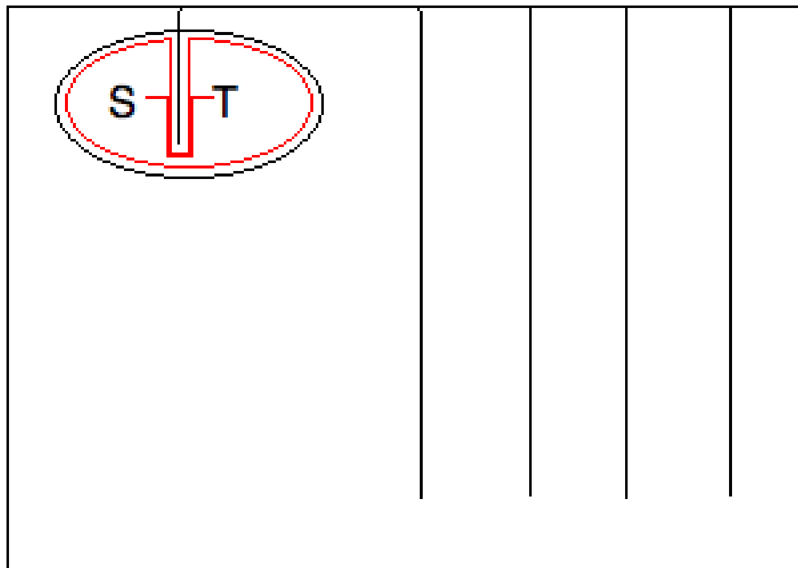


Figure 5: CBug improves upon Bug1. The red line indicates the path the robot will take.

2.2.1 Analysis of CBug

CBug improves upon Bug1 by preventing unnecessary obstacle traversal. An example of this is provided in Figure 5. The ellipse prevents the automaton from straying far from the target, unless it is necessary to reach the target, in which case the ellipse expands.

Gabriely and Rimon [2] do not analyze the performance of CBug by summing obstacle parameters, instead opting to compare it to the optimal path length, l_{opt}. l_{opt} is the best algorithmic solution to the given navigation problem, i.e. the offline solution. In other words, l_{opt} is the best possible path from the start to the target. An upper bound on the path length generated by CBug is proven to be $\frac{6\pi}{D}l_{opt}^2 + \|S-T\| + \frac{6A_0}{D}$ and a lower bound on the path length is proven to be $4\pi/(3D(1+\pi)^2) \times (1-\epsilon)l_{opt}^2$. In fact, the lower bound holds for all online algorithms in two dimensions, meaning it is a *universal* lower bound. The path length in both the upper and lower bound is of quadratic complexity. This relationship demonstrates that CBug is quadratically *competitive*, the essential claim of the paper.

2.2.2 Competitiveness

But what exactly does competitive mean? According to Gabriely and Rimon [2], competitiveness is "any functional relation between online performance and optimal off-line solution." Generally, competitiveness is the ability to relate the performance of an algorithm to the absolute best solution to the same problem. More specifically, Gabrielly and Rimon [2] also define *optimally competitive*, which means that the lower bound for all algorithms solving the problem is in the same complexity class as the upper bound of the algorithm being examined. Both CBug's upper bound and the universal lower bound are of the same complexity class (quadratic), so CBug is optimally competitive. This notion of competitiveness is a critical concept, and is also how the Boxes and CBoxes algorithms of Lucas Sabalka and Josh Brown Kramer [3] are analyzed.

2.3 Boxes and CBoxes

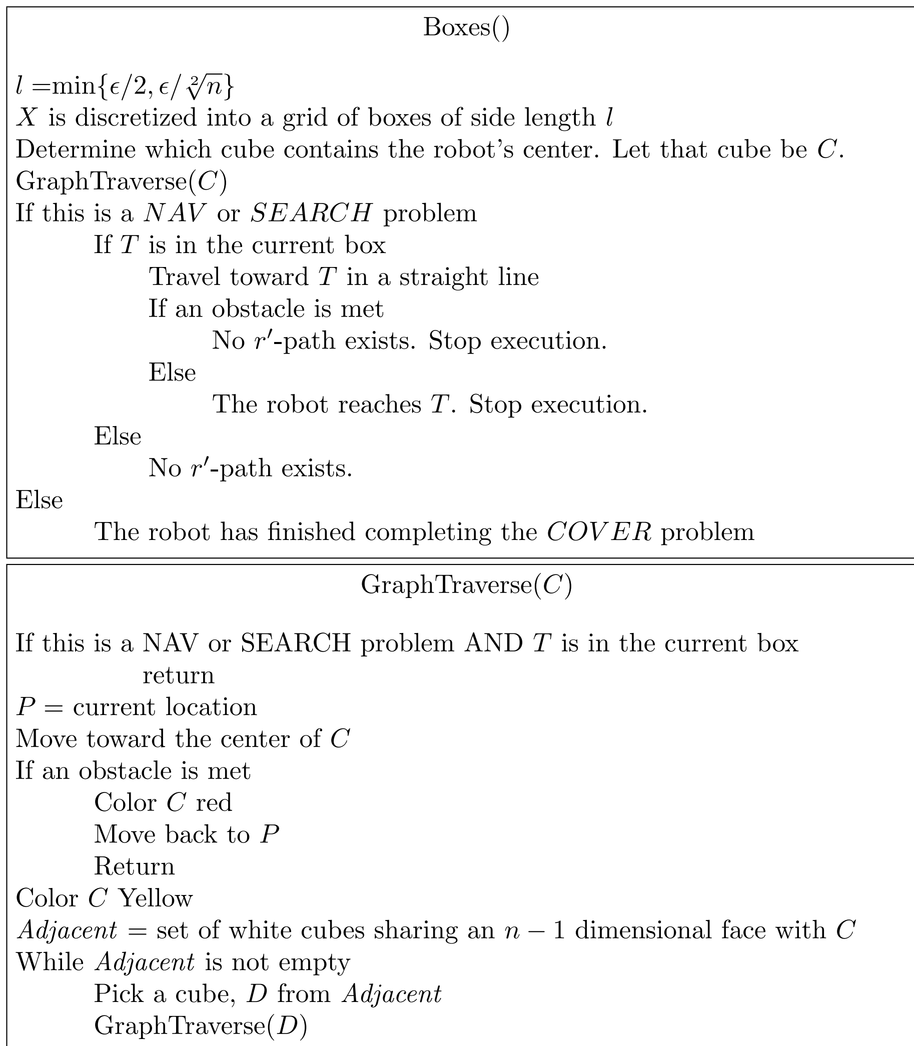
Josh Brown Kramer and Lucas Sabalka[3] add not only new robot motion tasks, but also consider higher dimensions. It is proven that there is no upper bound on path length for dimensions greater than or equal to three. This means that no algorithm can produce a path that can be proven to be bounded by a function of l_{opt}. However, a few modifications can allow the robot navigation tasks to be competitive. Brown Kramer and Sabalka choose to use a clearance parameter, ϵ, to solve this problem. ϵ can be as small as desirable and affects the paths the robot can take (more specifically, the tightness or narrowness of the paths the robot can take). In other words, the modification to the problem considers the case that the robot can avoid obstacles by at least ϵ. Brown Kramer and Sabalka then analyze the path length based upon these r'-paths, which is a path a robot of radius $r+\epsilon$ can take, and modify l_{opt} for a robot of radius r' as well.

The robot described in Brown Kramer and Sabalka's paper has a radius of $r > 0$ and is a sphere. The tasks that can be undertaken by the robot are COVER, SEARCH, and NAVIGATE. The COVER task is irrelevant to our research and will not be discussed and is only noted in the algorithms' pseudocode. SEARCH is the task in which the robot searches for a target, the location of which is not known. NAVIGATE, or NAV, requires the robot to reach a target, the location of which is known, unlike the SEARCH task. A nonspecific referral to one of these procedures is denoted by TASK. In this paper, we focus on the NAV task, but will make mention of the SEARCH task later. Lastly, let n denote the given dimension.

Before discussing the algorithms, we will discuss how the obstacle space is handled. The space is broken down into a cubical lattice where points in any given cube are at most ϵ apart. More simply, the space is sliced into a grid. The space is discretized to a graph and the algorithms perform a depth-first graph search of this space. Cubes or squares are colored differently to indicate different statuses. A pink cube is outside the virtual boundary, i.e. outside the ellipse. A red cube is a cube that indicates an obstacle is present that prevents the robot from reaching the center of the cube. A white cube is an unexplored cube and a yellow cube indicates a successfully explored cube. The coloring of cubes enables us to keep the knowledge of where the obstacles are present and avoid obstacle traversal, as the robot does not have to traverse entire obstacles like in Bug1 or CBug.

2.3.1 Boxes

Boxes starts by setting l as the lesser of $\epsilon/2$ and $\epsilon/\sqrt{n}$ and breaking the space into a cubical lattice, or grid, with all cubes white with side length l. Boxes then calls another algorithm, GraphTraverse, with input C (the current cube). We will describe GraphTraverse later in this paper. After GraphTraverse returns, Boxes checks to see if T is in the current box. If so, the robot will move toward T. If an obstacle is encountered, no r'-path exists and executions stops, as the cube in which T is present is colored red. Otherwise, the target is reached and executions stops. If T is not in the current cube, then no r'-path exists and executions stops. Pseudocode for Boxes and GraphTraverse is provided on the next page.



Boxes()

l =min$\{\epsilon/2, \epsilon/\sqrt[2]{n}\}$
X is discretized into a grid of boxes of side length l
Determine which cube contains the robot's center. Let that cube be C.
GraphTraverse(C)
If this is a NAV or $SEARCH$ problem
    If T is in the current box
        Travel toward T in a straight line
        If an obstacle is met
            No r'-path exists. Stop execution.
        Else
            The robot reaches T. Stop execution.
    Else
        No r'-path exists.
Else
    The robot has finished completing the $COVER$ problem

GraphTraverse(C)

If this is a NAV or SEARCH problem AND T is in the current box
        return
P = current location
Move toward the center of C
If an obstacle is met
    Color C red
    Move back to P
    Return
Color C Yellow
Adjacent = set of white cubes sharing an $n-1$ dimensional face with C
While *Adjacent* is not empty
    Pick a cube, D from *Adjacent*
    GraphTraverse(D)

2.3.2 GraphTraverse

GraphTraverse starts with a check whether T is in the current cube and returns if that is the case. It will then move the robot toward the center of the input cube. If an obstacle is encountered, it colors the cube it was moving to red, returns to the original position, and exits the function. If no obstacle is encountered, the cube is colored yellow. A set, *Adjacent*, is assembled consisting of the white cubes sharing an $n-1$ dimensional face with C, essentially all of the white neighbor cubes. While *Adjacent* is not empty, a cube, D, is selected from the set. GraphTraverse is called again, this time with input D. Pseudocode for GraphTraverse is provided above and a screenshot of Boxes running in 3D is provided in Figure 6.

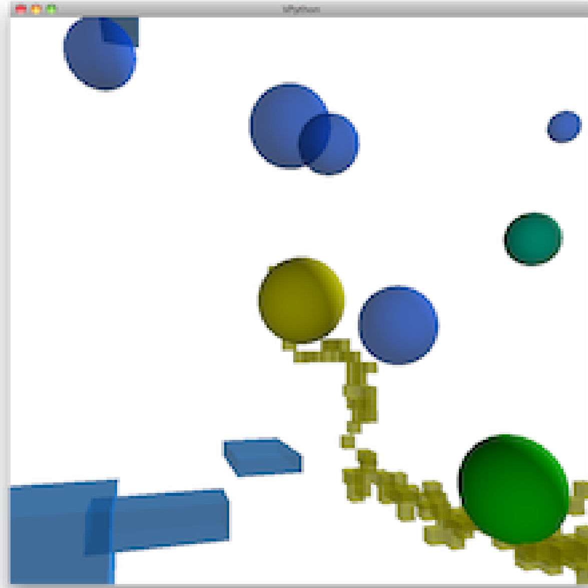


Figure 6: A screenshot of Boxes running in three dimensions.

2.3.3 Analysis of Boxes

Now for the analysis of Boxes. Boxes is similar to Bug1, as it operates without a bounding ellipse. However, Boxes adds the coloring of cubes and clearance parameter ϵ. Unlike Bug1, the robot is not required to trace an entire obstacle. Boxes is also distinct in that it can also use heuristics to select D in GraphTraverse.

It is proven by Brown Kramer and Sabalka [3] that Boxes finds an r-path if an r'-path exists. The lower bound is $c_n \frac{l^n_{opt}}{\kappa^{n-2}r'}$, where c_n is a constant dependent on n. However, no upper bound exists, and thus Boxes is not competitive.

2.3.4 CBoxes

We will now explain the CBoxes algorithm. It runs Boxes within an ellipse that expands when necessary, similar to how CBug ran Bug1 in an ellipse. The ellipse has foci at S and T for the NAV task, and both foci are at S for SEARCH (i.e. SEARCH is actually run in a sphere). Pseudocode for CBoxes is provided on the next page and a screenshot is provided in Figure 7.

CBoxes()

If the TASK is NAV
    $T' = T$
If the TASK is SEARCH
    $T' = S$
$a = d(S, T') + l$
While True
    Define the ellipsoid $\mathcal{E} = \{p : d(S, p) + d(p, T') \leq a\}$
    Color all cubes outside of $\mathcal{E}$ pink
    Call Boxes()
    If the robot is in the same cube as T
        If Bob is at T
            Task completed. Stop execution.
        Else
            No r'-path exists to T. Stop execution.
    Else if no neighbors of Pink cubes were explored
        No r'-path exists to T. Stop execution.
    Color all cubes white.
    $a = 2 \times a$.

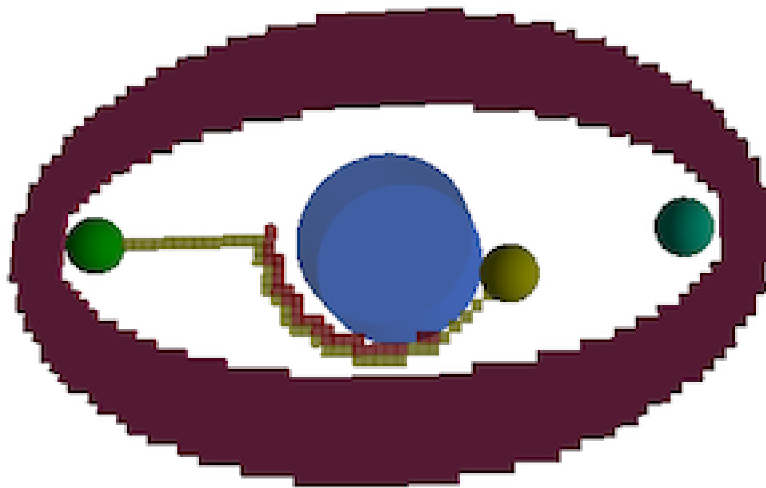


Figure 7: A screenshot of CBoxes running in 2D

2.3.5 Analysis of CBoxes

It is proven by Brown Kramer and Sabalka [3] that CBoxes will find r-path if an r'-path exists. As for the performance of CBoxes, the lower bound on path length of CBoxes is the same as that of Boxes, $c_n \frac{l_{opt}^n}{\kappa^{n-2} r'}$, where c_n is a constant dependent on n. The upper bound, is $c l_{opt}^n + d$ where d and c are dependent upon n and ϵ. We see that CBoxes is optimally competitive as both bounds are of the same complexity class.

3 Improvements

We took a Vpython implementation of Boxes running the NAV task and modified it. We essentially created CBoxes while adding multiple improvements to the algorithm.

3.1 User Interface

The initial user interface was inadequate. Users could not enter which dimension they would like the algorithm to run in, nor any parameters such as r. Furthermore, environments were entirely random, and obstacle courses could not be created or taken as input. Additionally, some improvements were already implemented, but could not be turned off.

Thus, we allowed the user to input parameters to the algorithm as well as control which improvements were running and which were not. We also implemented a system that allows the user to decide between a random environment, loading an environment, or creating their own. This made the program much more user-friendly and easy-to-use.

3.2 Taking Diagonals

This improvement was already implemented in the Boxes algorithm. This allows the robot to move to a cube sharing a face of arbitrary codimension, rather than just a codimension-1 face. More generally, this means the automaton can move diagonally. Brown Kramer and Sabalka [3] postulate this will worsen complexity estimates, but improve average case runtime by up to $\sqrt{n}$.

3.3 Greedy Heuristic

This is another improvement that was already implemented. Greedy applies to the selection of D in the Boxes algorithm. Essentially, all paths to T through white cubes (with the possible exception of end and start points) are calculated. D is then chosen according to the next cube on the shortest path. This heuristic allows the robot to avoid some obstacle traversal and head directly toward the target.

In our implementation, the user is asked whether he or she would like to run the greedy improvement. If not, then the selection of D is random among

all white cubes sharing a face of arbitrary codimension with the current cube. This essentially equates to changing the TASK from NAV to SEARCH, as the knowledge of T's location is not used.

3.4 Expansion of the Ellipsoid

Given that the original algorithm was Boxes, adding this improvement created CBoxes. To execute this function, all of the cubes completely outside of the ellipse are colored pink. However, the introduction of so many cube objects outside of the ellipse can affect runtime significantly. Thus, we chose not to make all of those cubes visible to the user. Instead, we make visible only the cubes $>$ the ellipse size and $<$ ellipseSize $+6 \times \epsilon$. This method makes it clear to the user that an ellipse is being used to contain the robot's path. $6 \times \epsilon$ was chosen because it can be proven that the pink cubes will form a connected ellipse when using $6 \times \epsilon$. All of the cubes outside the ellipse are colored pink in the robot's memory, thus not affecting the actual algorithm's performance, only the presentation to the user.

To implement this improvement, a list of all pink and yellow cubes is kept. Upon a call to the expandEllipse() function, all cubes in the list are deleted, removing all yellow and pink cubes from the display. Using for loops, the function runs through the robot's memory, changing any yellow or pink cubes to white. While this is occurring, cubes are being colored pink and displayed if all corners are $>$ the ellipse size and $<$ ellipseSize $+6 \times \epsilon$. All cubes outside of the ellipse are colored pink in the robot's memory. We also restrict the algorithm from coloring the newly colored pink cubes white, ensuring the function behaves properly.

3.5 Maximal Coloring

Maximal coloring takes advantage of some additional knowledge the robot has when contacting an obstacle. Normally, the robot colors only the cube he was trying to enter red when an obstacle is encountered. However, the robot actually knows that more cubes should be colored red as well. Since the cube is red, there is some point in the cube that is within r of an obstacle. This improvement essentially creates a sphere of radius r' with its center at the impact point, and colors all cubes completely within that sphere red.

The implementation of this is fairly simple. When an obstacle is encountered, the impact point (the point where the robot contacted the obstacle) is recorded. A function then iterates over the cubes close enough to be in the r' radius i.e. the $r+\epsilon$ radius of the impact point. If all corners of a cube are within r' distance of the impact point, the the cube is colored red.

The only complication that can occur with the maximal coloring improvement is that a yellow cube can be colored red. Under some circumstances, this could cause a false target unreachable message. However, by noting when a yellow cube is colored red, we can force the robot to backtrack. Once the robot

reaches a cube with a white neighbor cube, we stop the backtracking and allow the algorithm to proceed normally, thus avoiding the problem entirely.

3.6 Subdivision

The subdivision improvement works in tandem with the maximal coloring improvement. Subdivision allows cubes to be larger, thus allowing the robot to traverse the obstacle space more quickly. After coloring cubes via maximal coloring, any cubes not colored red are then subdivided. This means that the cube is broken down into 3^n cubes. The same process in maximal coloring is then used, only on the smaller, subdivided cubes: if all corners are within r' distance of the impact point, then they are colored red. Furthermore, if the subdivided cube has been previously visited, then it will be colored yellow. If a subdivided cube lies outside the virtual ellipse entirely, it will be colored pink. Subdivision allows the robot to take bigger steps toward the target, without a loss of precision for obstacle traversal.

This particular improvement was more difficult to implement. The original code represented the robot's memory with a 4-dimensional list, instead of objects. Because the indices of a list must be integers, it is impossible to store 9 different subdivided cubes in the original cube's position. Thus, we chose to keep a list of all subdivided cube objects and changed the color of a cube that has been subdivided in the robot's memory. We could access individual subcubes by searching the list for the cube's center position, as it is unique to each cube. We changed the color of the original cube to -1 in the robot's memory, which does not correspond to any other color (colors are stored as integers). This allows us to treat the cube like a white cube in calculating paths initially. The most difficult portion was pairing subdivision with the greedy heuristic, as the subdivided cubes must take on very similar values to those of their subdivided and non-subdivided neighbors. The implementation of that process is not perfect, but functional nevertheless. A perfect implementation of this improvement is a goal for future research.

4 Performance

4.1 Experiments

We generated obstacle courses with the newly created user interface. Thus, we could load the same environment and run the algorithm repeatedly, but with different improvements in effect. The obstacle courses used in the experiments were generated randomly. The size attributes of the obstacles were generated randomly from 1 to 15. However, the start and target points were kept at or near the same distance apart to keep the data uniform. Limits were placed upon the number of obstacles present in the environment, as too many obstacles produced a course in which the robot quickly determines it is trapped, and too few obstacles allowed the robot to quickly reach the target. We also kept the

obstacle positions within a certain range in order to keep them on the display and conceivably have an effect on the robot's path. In all cases, we measured path length running CBoxes in 2D.

4.2 Greedy Heuristic Data and Analysis

Table 1: Path lengths generated with and without Greedy improvement

	Greedy	Random	Difference	% Improvement
Env. 1	286.9	1071.10	784.41	73.23
Env. 2	295.93	1141.38	854.45	74.07
Env. 3	130.50	973.82	843.32	86.60
Env. 4	75.00	1096.28	1021.28	93.16
Env. 5	60.00	1190.85	1130.85	94.96
Average	169.62	1094.69	925.06	84.50

Given that running the algorithm without the greedy heuristic results in random movement, the data presented in Table 1 without the greedy heuristic is the average of five runs in the same environment. Environments four and five produce very low values for path length with the greedy improvement. This is due to the fact that these random courses happen to have few to no obstacles obstructing the greedy path. Thus, the path lengths generated for these cases with the greedy heuristic are very low. It is clear from the data that the greedy heuristic greatly improves algorithm performance in regards to path length. It is important to note, that given any obstacle course, greedy could perform worse than the alternative, due to randomness. These cases are exceedingly improbable and it is clear from the data that the greedy heuristic reduces path length in the vast majority of cases.

4.3 Maximal Coloring Data and Analysis

Table 2: Path lengths generated with and without maximal coloring

	Max Coloring	Without M.C.	Difference	% Improvement
Env. 1	144.61	198.72	54.11	27.22
Env. 2	152.37	198.09	45.72	23.08
Env. 3	123.69	123.38	-.31	-0.25
Env. 4	73.82	95.18	21.36	22.44
Env. 5	363.17	379.15	15.98	4.21
Env. 6	139.82	174.75	34.93	19.99
Env. 7	251.69	261.72	10.03	3.83
Env. 8	143.56	206.15	62.59	30.36
Average	174.09	204.64	30.55	14.93

Maximal coloring produces a small but noticeable improvement in general. Environment three is of particular interest, due to the maximal coloring improvement actually increasing the path length. In environment three, maximal coloring actually blocks off a path that can be taken when maximal coloring is not in effect. However, due to the advantage of traversing an obstacle using maximal coloring, it is still able to reach the target point in almost the same path length as operating without the improvement. There are cases such as these that could produce better performance without maximal coloring, but these are rare.

All of these tests were run with the greedy improvement in place. We anticipate that the difference with and without maximal coloring would actually be greater without the greedy heuristic in place. Running the algorithm sans the greedy heuristic results in longer path lengths and likely more obstacle traversal. Maximal coloring decreases obstacle traversal, and thus, intuitively, running maximal coloring without the greedy heuristic would yield a greater improvement over running the algorithm sans both maximal coloring and the greedy heuristic. Collecting data on this subject is another possible goal for future research.

4.4 Subdivision Data and Analysis

Table 3: Path lengths generated with and without subdivision and without maximal coloring

	Subdivision	No Sub.w/M.C.	No Sub., No M.C.
Env. 1	159.69	152.37	198.09
Env. 2	158.22	123.69	123.38
Env. 3	75.87	73.82	95.18
Env. 4	142.88	139.82	174.75
Env. 5	167.59	143.56	206.15
Average	140.85	126.65	159.51

The data indicates that subdivision actually increases path length in comparison to maximal coloring. Subdivision adds additional cubes that elongate the robot's path, potentially unnecessarily so. Furthermore, subdivision is only allowed to run in tandem with maximal coloring. Thus, subdivision does not produce the same benefits as maximal coloring and ends up elongating the path. However, subdivision still decreases path length in comparison to running the algorithm without subdivision and maximal coloring.

Again, these tests were run with greedy in place. Using the same notions as presented in the maximal coloring section, we anticipate the algorithm performance difference would be more drastic between subdivision and no maximal coloring when the greedy heuristic is not in place.

Upon observation of the algorithm being run with subdivision in effect, it is clear that subdivision decreases runtime. Intuitively, this occurs because

the robot travels to and from larger cubes and fewer movements. With fewer movements, calculations occur fewer times, speeding up the algorithm in terms of runtime.

5 Conclusions

We were successful in implementing the improvements in the algorithm. The user interface was drastically improved, allowing us to test and measure performance of the algorithm. We were also able to implement a method of running the algorithm without the greedy heuristic and enabled expansion of the ellipsoid, producing a program capable of running the CBoxes algorithm. Furthermore, we implemented maximal coloring and subdivision successfully.

Upon completion of the aforementioned improvements, we were able to measure the performance of the algorithm in different obstacle courses with different improvements in place. Our data shows that the greedy heuristic greatly improves algorithm performance (on average, almost 85%). Maximal coloring is also shown to improve algorithm performance, albeit a lesser improvement than the greedy heuristic. Subdivision does increase path length in comparison to maximal coloring, but still outperforms running the algorithm without maximal coloring. We also observe that subdivision improves runtime significantly. Overall, we accomplished our goals of implementing improvements in the algorithm and measuring algorithm performance in various situations.

6 Future Work

Much can be done to expand upon our research. First off, the entire code could be rewritten to a more simple implementation. For instance, as previously mentioned, the robot's memory is stored in a 4D list. Instead, one could make each cube an object, much like our implementation of a subdivided cube. This would make the code much more understandable and intuitive.

Another area in which the algorithm could be improved is the greedy heuristic for subdivided cubes. Our implementation is not perfect, sometimes relying on a straight-line heuristic, meaning the next cube could be determined by the Euclidean distance from the cube to the target. This can cause the robot to move into a subdivided cube unnecessarily, straying from the actual path the greedy heuristic would recommend. This fix would be slightly complicated, but is certainly feasible.

Subdivision could also be extended. In our implementation, cubes are only subdivided once. However, one could subdivide a cube numerous times, which would produce a more precise knowledge of obstacles. It is possible such an improvement could decrease path length, although our results indicate otherwise.

Lastly, more data could be acquired regarding the improvements. For instance, consider our comparison of max coloring and subdivision to the algorithm without the improvements. In all of those cases, the program was run

with the greedy heuristic in place. One could examine the performance without the greedy heuristic. As previously hypothesized, we suspect that the difference between the improved and original algorithm would be greater without using the greedy heuristic. Another idea is to use different heuristics for selecting the next cube and measure their performance against the greedy heuristic.

7 Acknowledgements

I'd like to thank Dr. Joshua Brown Kramer, my research advisor, for his instrumental guidance and aid in this research. I am very grateful for all the time and effort Dr. Brown Kramer spent with me on this thesis. I'd also like to thank Hans Joerg Tiede, my academic advisor and committee member. Lastly, I'd like to thank another committee member, Mark Liffiton. All of these professors have provided great assistance to me, and I could not be more appreciative.

8 Bibliography

[1] V. Lumelsky and A. Stepanov. Path planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape. Algorithmica, Volume 2, pages 403-430, 1987.

[2] Yoav Gabriely and Elon Rimon. CBug: A quadratically competitive mobile robot navigation algorithm. Preprint, January 2005.

[3] Joshua Brown Kramer and Lucas Sabalka. Multidimensional online robot motion. Preprint, November 2009.

[4] J.T. Schwartz and M. Sharir. On the "piano movers" problem: I. The case of a two-dimensional rigid polygonal boyd moving amidst polygonal barriers. Comm. Pure Appl. Math. 36, pages 345-398, 1983.

Appendices

A Python Code

```
#!/sw/bin/vpython2.4


# TO DO:
# create user interface
# allow environments to be specified by file
# fix behavior when a Yellow cube is colored Red.  Currently,
    Bob backtracks
#   even if that Yellow cube is not 'between' Bob and Start.
    Solution:
#   record cube path taken.
# rework how to compute shortest path to T for dimension 3.
# make coloring cubes Yellow also affect neighbors, so paths
    are through White cubes


from visual import *

from random import uniform
from random import choice
from visual.filedialog import save_file
from visual.filedialog import get_file

#######################################################
###                 Class Extensions                ###
#######################################################
#######################################################
# Extend the sphere, box, and cylinder classes to     #
# have an intersects method, computing whether two    #
# objects intesect (like Bob and an obstacle).        #
#                                                     #
#   WARNING: CYLINDER INTERSECTS METHOD IGNORES THE   #
#      Z-COORDINATE!  MADE FOR DIMENSION 2 CASE!      #
#######################################################
class subCube:

  def __init__(self, pos, color, tDistance):
    self.position = pos
    self.color = color
    self.tDistance = tDistance
    self.neighbors = []
  def getColor(self):
    return self.color
```

```
def makeNeighbors(self):
  a = [-1*l/3., 0, l/3.]
  if dimension == 2: c = [0]
  else: c = [-1*l/3., 0, l/3.]
  for m in a:
    for n in a:
      for o in c:
        #print "T"
        if not (m==0 and n==0 and o==0):
          #print "in if"
          location = self.position -[m,n,o]
          #print "COLOR CHECK: ", cubes.getColor(location)
          if cubes.getColor(location) == -1 :
            self.neighbors.append(location)
            #print "sub cube"
          else:
            new = getPos(getCoord(location))
            #print "check reg"
            if not self.alreadyIn(new):
              #print "reg cube"
              self.neighbors.append(new)
  #print "Actual position: ", self.position
  #for each in self.neighbors:
    #print "neighbor: ", each
def greedyApp(self):
  for c in self.neighbors:
    if cubes.getColor(c) == -1:
      sCube = subCubes.find(c)
      if sCube:
        if sCube.tDistance+1<self.tDistance:
          for each in sCube.neighbors: self.tDistance =
              sCube.tDistance + 1
    else:
      nCube = getCoord(c)
      if cubes.contents[nCube[0]][nCube[1]][nCube[2]][1] <
          self.tDistance:
        self.tDistance = cubes.contents[nCube[0]][nCube[1]][
            nCube[2]][1]

def allAreRed(self):
  for each in self.neighbors:
    if cubes.getColor(each) == -1:
      sCube = subCubes.find(each)
      if sCube.color != 3:
        return False
  return True
def listD(self, i, j, k):
  list = []
  if i < 0:
    if j < 0:
```

```
      s = subCubes.find(self.position+[l/3.,l/3., k])
      if s.color ==0: list.append(s.position)
      #if k < 0:
      #elif k = 0:
      #elif k > 0:
    elif j == 0:
      s = subCubes.find(self.position+[l/3.,l/3., k])
      t = subCubes.find(self.position+[l/3.,0, k])
      u = subCubes.find(self.position+[l/3.,-1*l/3., k])
      if s.color ==0: list.append(s.position)
      if t.color ==0: list.append(t.position)
      if u.color ==0: list.append(u.position)
      #if k < 0:
      #elif k = 0:
      #elif k > 0:
    elif j > 0:
      s = subCubes.find(self.position+[l/3.,-1*l/3., k])
      if s.color == 0: list.append(self.position+[l/3.,-1*l
          /3., k])
      #if k < 0:
      #elif k = 0:
      #elif k > 0:
  elif i == 0:
    if j < 0:
      s = subCubes.find(self.position+[0,l/3., k])
      t = subCubes.find(self.position+[-1*l/3.,l/3., k])
      u = subCubes.find(self.position+[l/3.,l/3., k])
      if s.color ==0: list.append(s.position)
      if t.color ==0: list.append(t.position)
      if u.color ==0: list.append(u.position)
      #if k < 0:
      #elif k = 0:
      #elif k > 0:
    elif j == 0:
      for each in self.neighbors:
        s = subCubes.find(each)
        if s.color == 0: list.append(s.position)
      #if k < 0:
      #elif k = 0:
      #elif k > 0:
    elif j > 0:
      s = subCubes.find(self.position+[-1*l/3.,-1*l/3., k])
      t = subCubes.find(self.position+[0,-1*l/3., k])
      u = subCubes.find(self.position+[l/3.,-1*l/3., k])
      if s.color ==0: list.append(s.position)
      if t.color ==0: list.append(t.position)
      if u.color ==0: list.append(u.position)
      #if k < 0:
      #elif k = 0:
      #elif k > 0:
```

```
      elif i > 0:
        if j < 0:
          s= subCubes.find(self.position+[-1*l/3.,l/3., k])
          if s.color == 0: list.append(s.position)
          #if k < 0:
          #elif k = 0:
          #elif k > 0:
        elif j == 0:
          s = subCubes.find(self.position+[-1*l/3.,-1*l/3., k])
          t = subCubes.find(self.position+[-1*l/3.,l/3., k])
          u = subCubes.find(self.position+[-1*l/3.,0, k])
          if s.color == 0: list.append(s.position)
          if t.color == 0: list.append(t.position)
          if u.color == 0: list.append(u.position)
          #if k < 0:
          #elif k = 0:
          #elif k > 0:
        elif j > 0:
          s = subCubes.find(self.position+[-1*l/3.,-1*l/3., k])
          if s.color ==0: list.append(s.position)
          #if k < 0:
          #elif k = 0:
          #elif k > 0:
      return list
#  def setTdistance(self, n): self.tDistance = n
#  def setColor(self, color): self.color = color
    def recolor(self, color):
      self.color = color
      if color == 1:
        a = box(pos = self.position,size=vector(l/3,l/3,l/3),
            color=vector(1,1,0), opacity = .5) # yellow
        pinkAndYellow.append(a)
      elif color == 2:
        a = box(pos = self.position,size=vector(l/3,l/3,l/3),
            color=vector(1,.3,.6), opacity = 1) # pink
        pinkAndYellow.append(a)
      elif color == 3: box(pos = self.position,size=vector(l/3,l
          /3,l/3),color=vector(1,0,0), opacity = .5) # red
#  def getCubes(self, coord): return neighbors
#  def getPos(self): return self.position
#  def setPos(self, pos): self.position = pos

    def alreadyIn(self, pos):
      delta = .0001
      for n in self.neighbors:
        if n[0] - pos[0] <= delta and n[0] - pos[0] >= -1*delta:
          if n[1] - pos[1] <= delta and n[1] - pos[1] >= -1*
              delta:
            if n[2] - pos[2] <= delta and n[2] - pos[2] >= -1*
                delta:
```

```
            return True
    return False

class SubList:
  def __init__(self): self.list = []
  def find(self, pos):
    #print "Looking for: ", pos
    delta = .0001
    for sCubes in self.list:
      #print "SubPOS: ", sCubes.position
      #print "Scolor: ", sCubes.color
      #print "DELTA: ", delta
      #print "First diff: ", sCubes.position[2] - pos[2]
      if sCubes.position[2] - pos[2] <= delta and sCubes.
          position[2] - pos[2] >= -1*delta:
        #print "z match - ", sCubes.position[0] - pos[0]
        if sCubes.position[0] - pos[0] <= delta and sCubes.
            position[0] - pos[0] >= -1*delta:
          #print "x match - ", sCubes.position[1] - pos[1]
          if sCubes.position[1] - pos[1] <= delta and sCubes.
              position[1] - pos[1] >= -1*delta:
            #print "y match"
            return sCubes
      if sCubes.position == pos:
        #print "in other"
        return sCubes
    return None
  def contains(self, pos):
    for sCubes in self.list:
      if sCubes.position == pos: return True
    return False
  def checkAllRed(self):
    count = 0
    index = 0
    l = []
    for sCubes in self.list:
      if sCubes.position == getPos(getCoord(sCubes.position)):
        if sCubes.color == 3:
          for each in sCubes.neighbors:
            s = self.find(each)
            l.append(s)
            if s.color == 3:
              count = count +1
          if count == 8 and dimension == 2:
            cubes.recolor(getCoord(sCubes.position), 3)
            for each in l:
              (self.list).remove(each)
            (self.list).remove(sCubes)
          if count == 26 and dimension == 3:
            cubes.recolor(getCoord(sCubes.position), 3)
```

```
          for each in l:
            (self.list).remove(each)
          (self.list).remove(sCubes)

def maxColor(self, impactPoint, possiblePos):
  pink = 0
  count = 0
  centerSubCube = self.find(possiblePos)
  b = [-1.*(1/6.), (1/6.)]
  if dimension == 2: c = [0]
  else: c = [-1.*(1/6.), (1/6.)]
  if centerSubCube.color == 0 or centerSubCube.color == 1: #
      not pink and not red
    for i in b:
      for j in b:
        for k in c:
          newPos = centerSubCube.position - [i,j,k]
          if mag(impactPoint-newPos) < r + epsilon: count =
              count + 1
          distance = mag(startPos - newPos) + mag(targetPos
              - newPos)
          if distance > ellipseSize and distance <=
              ellipseSize+epsilon*6: pink = pink + 1
    if (pink == 4): centerSubCube.recolor(2)
    if count == 4: centerSubCube.recolor(3)
    count = 0
    pink = 0
  for each in centerSubCube.neighbors:
    for i in b:
      for j in b:
        for k in c:
          newPos = each - [i,j,k]
          if mag(impactPoint-newPos) < r + epsilon: count =
              count + 1
          distance = mag(startPos - newPos) + mag(targetPos
              - newPos)
          if distance > ellipseSize and distance <=
              ellipseSize+epsilon*6: pink = pink + 1
    if (pink == 4):
      p = self.find(each)
      p.recolor(2)
    if count == 4:
      #print "each: ", each
      s = self.find(each)
      s.recolor(3)
    count = 0
    pink = 0
  if centerSubCube.allAreRed():
    cubes.recolor(getCoord(centerSubCube.position),3)
    #print "N", centerSubCube.neighbors
```

```
        for each in centerSubCube.neighbors:
          s = self.find(each)
          (self.list).remove(s)
          s.visible = false
          del s
        (self.list).remove(centerSubCube)
        centerSubCube.visible = false
        del centerSubCube
      cubes.countFromT()

class sphere(sphere):
    def intersects(self,a):
        if (type(a) == sphereType):    # intersects test for
            spheres
            return (mag(self.pos-a.pos) <= self.radius+a.
                radius)
        elif (type(a) == pointType):  # intersects test for
            sphere and point
            return (mag(self.pos-a) <= self.radius)
        elif (type(a) == boxType):     # intersects test for
            sphere and box
            distance=0                 # compute distance from
                sphere center to box
            cornerlo = a.pos - 0.5*a.size
            cornerhi = a.pos + 0.5*a.size
            center = self.pos
            for i in [0,1,2]:
                if   center[i] < cornerlo[i]: distance += (
                    center[i] - cornerlo[i])**2
                elif center[i] > cornerhi[i]: distance += (
                    center[i] - cornerhi[i])**2
                else: pass # this coordinate does not
                    contribute to distance
            if distance <= self.radius**2: return True
            else: return False
        else: pass

class box(box):
    def intersects(self,a):
        if (type(a) == sphereType):    # intersects test for
            box and sphere
            distance=0                 # compute distance from
                sphere center to box
            cornerlo = self.pos - 0.5*self.size
            cornerhi = self.pos + 0.5*self.size
            center = a.pos
            for i in [0,1,2]:
                if   center[i] < cornerlo[i]: distance += (
                    center[i] - cornerlo[i])**2
```

```
                elif center[i] > cornerhi[i]: distance += (
                    center[i] - cornerhi[i])**2
                else: pass            # this coordinate does
                    not contribute to distance
            if distance <= a.radius**2: return True
            else: return False
        elif (type(a) == pointType): # intersects test for
            box and point
            return self.intersects(sphere(pos=a,radius=0))
        elif (type(a) == boxType):   # intersects test for
            boxes
            acornerlo = a.pos - 0.5*a.size
            acornerhi = a.pos + 0.5*a.size
            scornerlo = self.pos - 0.5*self.size
            scornerhi = self.pos + 0.5*self.size
            for i in [0,1,2]:          # test by coordinate if
                it's okay
                if   acornerhi[i] < scornerlo[i]: return False
                elif acornerlo[i] > scornerhi[i]: return False
                else: pass            # this coordinate is
                    okay
            return True
        else: pass

    def getPos(self):
        return vector(self.pos)

class cylinder(cylinder):
    def intersects(self,a):
        #WARNING: IGNORES Z-COORDINATE! MADE FOR DIMENSION
            2!
        selfp = vector(self.pos)
        selfp[2] = 0
        p = vector(a.pos)
        p[2] = 0
        if (type(a) == sphereType):   # intersects test for
            spheres
            return (mag(selfp-p) <= self.radius+a.radius)
        elif (type(a) == pointType): # intersects test for
            sphere and point
            return (mag(selfp-p) <= self.radius)
        elif (type(a) == boxType):   # intersects test for
            sphere and box
            distance=0               # compute distance from
                sphere center to box
            cornerlo = p - 0.5*a.size
            cornerhi = p + 0.5*a.size
            center = self.pos
            for i in [0,1]:
```

```
                if   center[i] < cornerlo[i]: distance += (
                    center[i] - cornerlo[i])**2
                elif center[i] > cornerhi[i]: distance += (
                    center[i] - cornerhi[i])**2
                else: pass             # this coordinate does
                    not contribute to distance
            if distance <= self.radius**2: return True
            else: return False
        else: pass




########################################################
###                    Object Types                   ###
########################################################
########################################################
#  * boxType                                           #
#  * cylinderType                                      #
#  * ellipsoidType                                     #
#  * sphereType                                        #
#  * pointType                                         #
########################################################
A = box()
boxType = type(A)
A.visible=0
A = cylinder()
cylinderType = type(A)
A.visible=0
A = ellipsoid()
ellipsoidType = type(A)
A.visible=0
A = sphere()
sphereType = type(A)
A.visible=0
A = vector(0,0,0)
pointType = type(A)
A.visible=0
pinkAndYellow = []

########################################################
###                  Global Variables                 ###
########################################################
########################################################
#  * r                                                 #
#  * epsilon                                           #
#  * l                                                 #
#  * t                                                 #
#  * dt                                                #
```

```
#  * dimension                                                    #
#  * startPos                                                     #
#  * targetPos                                                    #
#  * centerPos                                                    #
#  * maxCoord                                                     #
#  * centerCoord                                                  #
#  * startCoord                                                   #
#  * targetCoord                                                  #
#################################################################

#global printing
#global moves
m = None
e = 0
d = None
a = None
moves = 0
dMove = 0
subMove = 0
dSubMove = 0
rTemp = 0
sTemp = 0
tTemp = 0
g = None
s = None
#printing = true
printing = false
while not (d == "2" or d == "3"):
  d = raw_input("Please enter the desired dimension: ")
  if d == "2":
    dimension = 2
  elif d == "3":
    dimension = 3
  else:
    print "Please enter a dimension 2 or 3."

while not (a == "1" or a == "0"):
  a = raw_input("Press 1 for CBoxes and 0 for Boxes: ")
  if a == "1": ellipseToggle = True
  elif a == "0": ellipseToggle = False

while not (m == "1" or m == "0"):
  m = raw_input("Turn Maximal Coloring on? (Enter 1 for yes, 0
      for no): ")
  if m == "1": maxColor = True
  elif m == "0":
    maxColor = False
    subD = False

if (maxColor):
```

```
  while not (s == "1" or s == "0"):
    s = raw_input("Turn subdivision on? (Enter 1 for yes, 0 for no): ")
    if s == "1": subD = True
    elif s == "0": subD = False

while not (g == "1" or g =="0"):
  g = raw_input("Turn Greedy on? (Enter 1 for yes, 0 for no): ")
  if g == "1": greedy = True
  elif g == "0": greedy = False

if ellipseToggle:
  while not(float(e) <= 2.0 and float(e) > 1.0):
    e = raw_input("By what factor would you like the ellipse to expand? (1-2): ")
    expand = float(e)

while not(float(rTemp) > 0):
  rTemp = raw_input("What is Bob's radius? (r > 0): ")
  r = float(rTemp)
#epsilon = float(r)/10
#l = r
#epsilon = 2*r
epsilon = r/((dimension)**(1./2))
if (epsilon/(dimension**(1./2)) > epsilon/2):
  l = epsilon/2
else:
  l = epsilon/(dimension**(1./2))
t = 1
if (subD):
  oldL = l
  l = 3*l#r/(2*((dimension)**(1./2)))
print l
print epsilon

sX = raw_input("Enter the x coordinate of the start: ")
sY = raw_input("Enter the y coordinate of the start: ")
if dimension == 3:
  sZ = raw_input("Enter the z coordinate of the start: ")
  tZ = raw_input("Enter the z coordinate of the target: ")
else:
  sZ = 0
  tZ = 0
tX = raw_input("Enter the x coordinate of the target: ")
tY = raw_input("Enter the y coordinate of the target: ")
# Initialize default start and target positions
startPos  = vector(int(sX),int(sY),int(sZ))
targetPos = vector( int(tX), int(tY), int(tZ))
```

```
centerPos = (startPos+targetPos)/2


maxCoord     = int(floor(1.5*mag(startPos-targetPos)/l)+1)
if dimension == 2:
    centerCoord = vector(int(floor(maxCoord/2)),int(floor(
        maxCoord)/2),0)
if dimension == 3:
    centerCoord = vector(int(floor(maxCoord/2)),int(floor(
        maxCoord)/2),int(floor(maxCoord)/2))




# set the camera direction and range
if dimension == 2:
    scene.forward = vector(.1,.3,1)
    scene.range = (75,75,75)
if dimension == 3:
    scene.forward = vector(1,.1,.3)
    scene.range = (20,20,20)
scene.background = (1,1,1)
scene.visible = False
scene.height = 800
scene.width = 800
scene.visible = True

########################################################
###                  Math Functions                  ###
########################################################
########################################################
#  * getCoord(v)                                       #
#  * getPos(coordv)                                    #
#  * normalize(v)                                      #
########################################################


# Return the grid coordinates of the cube containing
# the point v
def getCoord(v):
    cubeCoord = range(3)
    for i in range(0,3):
        cubeCoord[i] = int(centerCoord[i]-floor((centerPos[i]-
            v[i])/l+.5))
    return cubeCoord


# Return the vector corresponding to the center of the
# cube with the given coordinates
def getPos(coord):
```

```
    cubePos = range(3)
    for i in range(0,3):
        cubePos[i] = centerPos[i] - l*(centerCoord[i]-coord[i
            ])
    return vector(cubePos)


# Return the vector of magnitude 1 in the direction
# of someVector
def normalize(v):
    if mag(v) == 0: return v
    temp = v
    for i in range(0,3):
        temp[i] /= mag(v)
    return temp



def binaryString(i):
    b = ': ' + str(i)
    if dimension == 2:
        i >>= 9
    while i > 0:
        j = i & 1
        b = str(j) + b
        i >>= 1
    return b

def dList(v, pos):
  list = []
  i = v[0]
  j = v[1]
  k = v[2]
  #print "i ", i
  #print "j ", j
  #print "k ", k
  #print "V", v
  #print "POS: ", pos
  if i < 0:
    if j < 0:
      s = subCubes.find(pos+[l/3.,l/3., k])
      #print "s.color", s.color
      if s.color ==0: list.append(s.position)
      #if k < 0:
      #elif k = 0:
      #elif k > 0:
    elif j == 0:
      s = subCubes.find(pos+[l/3.,l/3., k])
      t = subCubes.find(pos+[l/3.,0, k])
      u = subCubes.find(pos+[l/3.,-1*l/3., k])
```

```
    if s.color ==0: list.append(s.position)
    #print "L1 ", list
    if t.color ==0: list.append(t.position)
    #print "L2 ", list
    if u.color ==0: list.append(u.position)
    #print "L3 ", list
    #if k < 0:
    #elif k = 0:
    #elif k > 0:
  elif j > 0:
    s = subCubes.find(pos+[l/3.,-1*l/3., k])
    if s.color == 0: list.append(pos+[l/3.,-1*l/3., k])
    #if k < 0:
    #elif k = 0:
    #elif k > 0:
elif i == 0:
  if j < 0:
    s = subCubes.find(pos+[0,l/3., k])
    t = subCubes.find(pos+[-1*l/3.,l/3., k])
    u = subCubes.find(pos+[l/3.,l/3., k])
    if s.color ==0: list.append(s.position)
    #print "L1 ", list
    if t.color ==0: list.append(t.position)
    #print "L2 ", list
    if u.color ==0: list.append(u.position)
    #print "L3 ", list
    #print "S", s.color
    #print "T", t.color
    #print "U", u.color
    #if k < 0:
    #elif k = 0:
    #elif k > 0:
  elif j == 0:
    for each in sCube.neighbors:
      if each.color == 0: list.append(each.position)
    #if k < 0:
    #elif k = 0:
    #elif k > 0:
  elif j > 0:
    s = subCubes.find(pos+[-1*l/3.,-1*l/3., k])
    t = subCubes.find(pos+[0,-1*l/3., k])
    u = subCubes.find(pos+[l/3.,-1*l/3., k])
    if s.color ==0: list.append(s.position)
    if t.color ==0: list.append(t.position)
    if u.color ==0: list.append(u.position)
    #if k < 0:
    #elif k = 0:
    #elif k > 0:
elif i > 0:
  if j < 0:
```

```
      s= subCubes.find(pos+[-1*l/3.,l/3., k])
      if s.color == 0: list.append(s.position)
      #if k < 0:
      #elif k = 0:
      #elif k > 0:
    elif j == 0:
      s = subCubes.find(pos+[-1*l/3.,-1*l/3., k])
      t = subCubes.find(pos+[-1*l/3.,l/3., k])
      u = subCubes.find(pos+[-1*l/3.,0, k])
      if s.color == 0: list.append(s.position)
      if t.color == 0: list.append(t.position)
      if u.color == 0: list.append(u.position)
      #if k < 0:
      #elif k = 0:
      #elif k > 0:
    elif j > 0:
      s = subCubes.find(pos+[-1*l/3.,-1*l/3., k])
      if s.color ==0: list.append(s.position)
  #print "L: ", list
  return list

startCoord  = getCoord(startPos)
targetCoord = getCoord(targetPos)
totalDistance = 0
start  = sphere(pos=startPos , color=(0,1,0), radius=r)
target = sphere(pos=targetPos, color=(0,1,.8), radius=r)
Bob    = sphere(pos=startPos , color=(1,1,0), radius=r)


########################################################
###                 Class Definitions                ###
########################################################
########################################################
# In this section, we define:                          #
# * the Obstacles class, used to store obstacle sets   #
# * the Cubulation class, used to store the            #
#   subdivision grid and colors of cubes               #
########################################################



# The Obstacles class defines an object which contains
# an arbitrary number of obstacles. Each obstacle is
# added using the Obstacles.add command. One can also
# test for whether Bob intersects an obstacle using
# the Obstacles.intersects test. Random spheres and
# boxes may also be added using addRandom*** methods.
class Obstacles:
    contents = []
    def add(self,a):
```

```
        self.contents.append(a)
    def intersects(self,bot):
        for thing in self.contents:
            if thing.intersects(bot): return True
        return False
    # We want to be able to add a bunch of boxes as obstacles
    def addRandomBox(self):
        a = range(0,6)
        toReturn = ""
        a[0] = uniform(-50,50)
        a[1] = uniform(-50,50)
        a[2] = uniform(-50,50)
        for j in range(3,6): a[j] = uniform(0,15)
        if dimension == 2:
      a[2] = 0
      a[5] = 10
        p = a[:3]
        s = a[3:]
        b = box(pos=p, size=s, color=(0,.5,1), opacity = .5)
        #b = box(pos=a[:3], size=a[3:], color=(0,.5,1),
            opacity = .5)
        self.add(b)
        toReturn = str(a[0]) + "␣" + str(a[1]) + "␣" + str(a
            [2]) + "␣b␣" + str(s[0]) + "␣" + str(s[1]) + "␣" +
             str(s[2]) + "\n"
        return toReturn
    # Let's be able to add random spheres too
    def addRandomSphere(self):
        randomp = range(0,3)
        randomp[0] = uniform(-50,50)
        randomp[1] = uniform(-50,50)
        randomp[2] = uniform(-50,50)
        r = uniform(1,8)
        self.add( sphere( pos=randomp, radius = r, color =
            (0,.3,1), opacity = .5 ) )
        toWrite = str(randomp[0]) + "␣" + str(randomp[1]) + "␣
            " + str(randomp[2]) + "␣s␣" + str(r) + "\n"
        return toWrite
        #self.add( sphere( pos=randomp, radius = uniform(1,8),
             color = (0,.3,1), opacity = .5 ) )
    # and cylinders when dimension == 2
    def addRandomCylinder(self):
        randomp = range(0,3)
        randomp[0] = uniform(-50,50)
        randomp[1] = uniform(-50,50)
        randomp[2] = -4.5
        r = uniform(1,8)
        self.add(cylinder(pos=randomp,axis=(0,0,9),radius=r,
            opacity = .5, color = (0,.3,1)))
```

```
        #self.add(cylinder(pos=randomp, axis=(0,0,9), radius=
            uniform(1,8), opacity = .5, color = (0,.3,1)))
        toWrite = str(randomp[0]) + " " + str(randomp[1]) + " 
            " + str(randomp[2]) + " c " + str(r) + "\n"
        return toWrite


# CubeArray class is used to store Bob's memory.  It contains
    an array of
# boxes, called contents.  The structure of contents is an
    array, where
# contents[x][y][z] records information about the cube with
    center
# coordinate centerPos+l(x,y,z).  Each cube has 5 attributes.
# 0:  The 0 attribute is color:
#    0 = White
#    1 = Yellow
#    2 = Pink
#    3 = Red
# 1:  The 1 attribute is distance to T through White and
    Yellow cubes.
# 2:  The 2 attribute indicates which neighbors of the cube
    are exactly
# one step closer to T.
# 3:  The 3 attribute indicates which neighbors of the cube
    are exactly
# one step farther from T.
# 4:  The 4 attribute records which messages from T have
    reached the
# cube, and is used for computing distance to T.
#
# The 2 and 3 attributes are recorded via bits, with each
# neighbor getting one bit.  If the bit is 1 (on), the
    neighbor is
# exactly one step closer to/farther from T; otherwise, the
    bit is
# 0 (off).  The $i^{th}$ bit of the integer -- corresponding
    to
# $2^{i-1}$ -- represents the $i^{th}$ neighbor in the reverse
# lexicographic ordering of the neighbors.  Thus, if
# $a,b \in \{-1,0,1\}$, then neighbor [x+a][y+b][z+c]
    corresponds to
# index
#    $$1*(a+1)+ 3*(b+1) + 9(c+1).$$
# Index $13$ -- corresonding to $[x][y][z]$ -- of each
    attribute is
# always 0.
#
# If dimension == 2, note z and c are always 0.
```

```
#
# CubeArray also contains a number of methods:
#
#    PRIVATE METHODS:
#    arg(self,argument,coord):
#            Deals with the structure of contents[].
#    influences(self,coord,closefar):
#            Used to calculate the (important) neighbors of a
   cube.  If
#            closefar is 2, the neighbors are closer to T; if
   it's 3, farther.
#    getIndex(self,currentCoord,neighborCoord):
#            Deals with the structure of contents[] by
   returning index of
#            the appropriate bit corresponding to the
   relationship
#            between currentCoord and neighborCoord.
#    notifyNeighbors(self,coord,position):
#            Tell neighbors coord is now Red.
#
#    PUBLIC METHODS:
#    recolor(self, coord, newcolor):
#            Records new colors and generates visible cubes.
#    getColor(self,position):
#            Returns the color of the cube containing position.
#    addObstacle(self,position):
#            Add an obstacle point, and all the colorings and
#            calculations that result.
#    countFromT(self):
#            Label each cube with its distance through White
   cubes to T.
#    bestD(self,position):
#            Return the coordinates of an adjacent cube on a
   geodesic to
#            T which is not Yellow.
#    expandEllipse(self):
#            Expands virtual ellipsoid.  Currently undeveloped.
class CubeArray:
    cubeCount = 0
    if dimension == 2:
        contents = [[[[0,0,0,0,0]] for i in range(maxCoord)]
            for i in range(maxCoord)]
    if dimension == 3:
        contents = [[[[0,0,0,0,0] for i in range(maxCoord)]
            for i in range(maxCoord)] for i in range(maxCoord)
            ]

#######PRIVATE METHODS########
    def arg(self,argument,coord):
```

```
        return self.contents[coord[0]][coord[1]][coord[2]][
            argument]

#Returns a list of all cubes closer to (when closefar == 2) or
     farther from
#(when closefar ==3) T.
    def influences(self,coord,closefar):
        #if coord is unreachable from T through White cubes,
            return
        if self.arg(4,coord) != self.arg(4,targetCoord):
            return []
        theyAre = []
        bitString = self.arg(closefar,coord)
        for index in range(0,27):
            if bitString&(1<<index):
                theyAre.append([int(coord[0]-1+(index)%3),int(
                    coord[1]-1+(index//3)%3),int(coord[2]-1+(
                    index//9)%9)])
                neighborCoord = [int(coord[0]-1+(index)%3),int
                    (coord[1]-1+(index//3)%3),int(coord[2]-1+(
                    index//9)%9)]
        return theyAre


    def getIndex(self,currentCoord,neighborCoord):
        return 1*(neighborCoord[0]-currentCoord[0]+1) + 3*(
            neighborCoord[1]-currentCoord[1]+1) + 9*(
            neighborCoord[2]-currentCoord[2]+1)

    def printMemory(self):
        for i in range(targetCoord[0]+3,startCoord[0]-3,-1):
            for j in range(targetCoord[1]+3,startCoord
                [1]-3,-1):
                if self.arg(0,[j,i,0]) > 1:
                    if mag(vector(getCoord(Bob.pos)) - (j,i,0)
                        ) < 1:
                        print "X",
                    else:
                        print "*",
                elif mag(vector(getCoord(Bob.pos)) - (j,i,0))
                    < 1:
                    print "B",
                elif mag(vector(startCoord)-(j,i,0)) < 1:
                    print "S",
                elif mag(vector(targetCoord)-(j,i,0)) < 1:
                    print "T",
                elif self.arg(0,[j,i,0]) == 1:
                    print "Y",
                else:
                    print (self.arg(1,[j,i,0]))%10,
```

```
        print ''

    def findNewNeighbor(self, initial):            #recalculates
        influences locally
        needRedo = False

        affectedList = [initial]
        self.contents[initial[0]][initial[1]][initial[2]][0]
            += 4

        for coord in affectedList:
            for neighbor in self.influences(coord,3):
                neighborArgs = self.contents[neighbor[0]][
                    neighbor[1]][neighbor[2]]
                neighborArgs[2] -= (1<<self.getIndex(neighbor,
                    coord))
                if neighborArgs[2] == 0:
                    neighborArgs[0] += 4            #temporarily
                        color puce or yellowy puce
                    affectedList.append(neighbor)
            self.contents[coord[0]][coord[1]][coord[2]][3] = 0
        if len(affectedList) > 150:                #if there are
            too many Puce cubes,
                                                   #just re-
                                                       countFromT
            for coord in affectedList:
                self.contents[coord[0]][coord[1]][coord[2]][0]
                    -= 4
            return True

        while affectedList != []:
            tempList = []                          #stores coord,
                bestdist, bestneighbors
            for coord in affectedList:
                bestdist = self.arg(1,startCoord)+10
                bestneighbors = []
                for i0 in range(-1,2):
                    for i1 in range(-1,2):
                        for i2 in range(2-dimension,dimension
                            -1):
                        #for every neighbor cube
                            neighborCoord = [coord[0]+i0,coord
                                [1]+i1,coord[2]+i2]
                            if neighborCoord[0] < maxCoord and
                                neighborCoord[1] < maxCoord
                                and neighborCoord[2] <
                                maxCoord:
                neighborArgs = self.contents[neighborCoord
                    [0]][neighborCoord[1]][neighborCoord[2]]
```

```
    if neighborArgs[0] == 0:
    #if neighbor is White
        if neighborArgs[1]+1 < bestdist:
            bestdist = neighborArgs[1]+1
            bestneighbors = [neighborCoord]
        elif neighborArgs[1]+1 == bestdist:
            bestneighbors.append(neighborCoord)
    tempList.append([coord,bestdist,bestneighbors
        ])

bestElement = tempList[0]
for element in tempList:
    if element[1] < bestElement[1]:
        bestElement = element
bestCoord = bestElement[0]
bestArgs = self.contents[bestCoord[0]][bestCoord
    [1]][bestCoord[2]]

bestArgs[0] -= 4
bestArgs[1] = bestElement[1]
for bestNeighbor in bestElement[2]:
    self.contents[bestNeighbor[0]][bestNeighbor
        [1]][bestNeighbor[2]][3] += (1<<self.
        getIndex(bestNeighbor,bestCoord))
    bestArgs[2] += (1<<self.getIndex(bestCoord,
        bestNeighbor))
for i0 in range(-1,2):
    for i1 in range(-1,2):
        for i2 in range(2-dimension,dimension-1):
        #for every neighbor cube
            neighborCoord = [bestCoord[0]+i0,
                bestCoord[1]+i1,bestCoord[2]+i2]
            if neighborCoord[0] < maxCoord and
                neighborCoord[1] < maxCoord and
                neighborCoord[2] < maxCoord:
  neighborArgs = self.contents[neighborCoord[0]][
      neighborCoord[1]][neighborCoord[2]]
  if neighborArgs[0] < 2 and neighborArgs[1] ==
      bestArgs[1]+1:
  #if neighbor is exactly one step farther, add
      influence
    neighborArgs[2] += (1<<self.getIndex(
        neighborCoord,bestCoord))
    bestArgs[3] += (1<<self.getIndex(bestCoord,
        neighborCoord))
if bestElement[2] == []:
    needRedo = True

affectedList.remove(bestCoord)
```

```
        return needRedo


    def notifyNeighbors(self,coord,position):       # tell
        neighbors coord is now Red
    #print "NOTIFY NEIGHBORS"
    if self.arg(2,coord) - 2 == 0:
      return False
    needRedo = False
    for neighbor in self.influences(coord,3): # those further
        from T
      if (neighbor[0] < maxCoord and neighbor[1] < maxCoord
          and neighbor[2] < maxCoord):
        self.contents[neighbor[0]][neighbor[1]][neighbor
            [2]][2] -= (1<<self.getIndex(neighbor,coord))
      if (coord[0] < maxCoord and coord[1] < maxCoord and
          coord[2] < maxCoord):
        self.contents[coord[0]][coord[1]][coord[2]][3] -= (1<<
            self.getIndex(coord,neighbor))
      if int(self.arg(2,neighbor)) == 0:
        if self.findNewNeighbor(neighbor):
          return True
    for neighbor in self.influences(coord,2): # those closer
        to T
      if (neighbor[0] < maxCoord and neighbor[1] < maxCoord
          and neighbor[2] < maxCoord and self.contents[
          neighbor[0]][neighbor[1]][neighbor[2]][0] != 2):
        self.contents[neighbor[0]][neighbor[1]][neighbor
            [2]][3] -= (1<<(self.getIndex(neighbor,coord)))
      if (coord[0] < maxCoord and coord[1] < maxCoord and
          coord[2] < maxCoord):
        self.contents[coord[0]][coord[1]][coord[2]][2] -= (1<<
            self.getIndex(coord,neighbor))
    return needRedo



#######PUBLIC METHODS########
    def recolor(self, coord, newcolor):
        self.contents[coord[0]][coord[1]][coord[2]][0] =
            newcolor
        if newcolor - 1 == 0:                          # Yellow
            self.cubeCount += 1
            b = box(pos = getPos(coord),size=vector(1,1,1),
                color=vector(1,1,0), opacity = .5)
            pinkAndYellow.append(b)
        elif newcolor - 2 == 0:                        # Pink
            self.cubeCount += 1
            b= box(pos=getPos(coord),size=vector(1,1,1),color=
                vector(1,.7,.7), opacity = .5) # 1 .3 0
```

```
            pinkAndYellow.append(b)
        elif newcolor - 3 == 0:                          # Red
            self.cubeCount += 1
            b = box(pos=getPos(coord),size=vector(l,l,l),color
                =vector(1,0,0), opacity = .5)
            #boxArray.append(b)

    def getColor(self, position):
        return self.arg(0,getCoord(position))

    def addObstacle(self, position, next):

    # note that position is the intersection point, or point
        of impact
    # next is the cube center that Bob was trying to move to

        maxCoordDiff = int(floor((r+epsilon)/l)) # ? 1 as l =
            r
        coord = getCoord(position) # coord of point of impact
        needRedo = False
        needReturn = False
        if self.arg(0,getCoord(next)) != -1: self.recolor(
            getCoord(next),3) # this colors the cube we were
            moving to red
        else:
      w = subCubes.find(next)
      w.recolor(3)
      #print "recolored where we were headed: ", w.position
        if self.arg(0,getCoord(next)) == 1 or self.
            notifyNeighbors(getCoord(next),position):
    # next is yellow? or ???
            needRedo = True
        a = [ -1.*(l/2.), (l/2.)]
        count = 0
        if (dimension == 2):
        for i1 in range(-maxCoordDiff,maxCoordDiff+1): # -1 to
             2
          for i2 in range(-maxCoordDiff,maxCoordDiff+1): # -1
              to 2
          possibleCoord = [int(coord[0]+i1),int(coord[1]+i2),
              int(coord[2])]
          possiblePos = getPos(possibleCoord)
          possibleColor = self.arg(0,possibleCoord)
          if (possibleColor != 3 and possibleColor != -1): #
              no use checking if it is already red
            for j in a:
              for k in a:
                if mag(position-possiblePos-[j,k,0]) < r+
                    epsilon: # not red and in r neighborhood
                  count = count + 1
```

```
        if count == 4:
          self.recolor(possibleCoord,3) # color it red
          if self.notifyNeighbors(possibleCoord,position):
            needRedo = True
          if possibleColor == 1: # yellow
            needReturn = True
            returnToCoord = possibleCoord
        elif (subD and possibleColor == 0): self.
            subdivision(possiblePos, position)
        count = 0
      elif possibleColor == -1: subCubes.maxColor(position
          , possiblePos)
    if (dimension == 3):
    for i1 in range(-maxCoordDiff,maxCoordDiff+1): # -1 to
        2
      for i2 in range(-maxCoordDiff,maxCoordDiff+1): # -1
          to 2
        for i3 in range(-maxCoordDiff,maxCoordDiff+1): #
            -1 to 2 for 3d
        possibleCoord = [int(coord[0]+i1),int(coord[1]+i2)
            ,int(coord[2]+i3)]
        possiblePos = getPos(possibleCoord)
        possibleColor = self.arg(0,possibleCoord)
        if possibleColor != 3: # no use checking if it is
            already red
          for j in a:
            for k in a:
              for m in a:
                if mag(position-possiblePos-[j,k,m]) < r +
                      epsilon: # not red and in r
                    neighborhood
                  count = count + 1
          if count == 8:
              self.recolor(possibleCoord,3) # color it red
              if self.notifyNeighbors(possibleCoord,
                  position):
              needRedo = True
              if possibleColor == 1: # yellow
              needReturn = True
              returnToCoord = possibleCoord
          elif (subD):
            self.subdivision(possiblePos, position)
          count = 0
    if needReturn:
  return [-1,returnToCoord]
    if needRedo:
  self.countFromT()
    return [0]

def subdivision(self, centerPosition, impactPoint):
```

```
global l
count = 0
pink = 0
new = []
a = [ -1.*(l/3.), 0, (l/3.)]
b = [-1.*(l/6.), (l/6.)]
if (dimension == 2):
  for i1 in a:
    for i2 in a:
      possiblePos = centerPosition -[i1 ,i2 ,0]
      possibleColor = self.arg(0,getCoord(possiblePos))
      possibleDist = self.arg(1, getCoord(possiblePos))
      for j in b:
        for k in b:
          if mag(impactPoint-possiblePos -[j ,k ,0]) < r +
              epsilon: #in r neighborhood
            count = count + 1
          distance = mag(startPos - possiblePos - [j ,k ,0])
               + mag(targetPos - possiblePos - [j ,k ,0])
          if distance > ellipseSize and distance <=
              ellipseSize+epsilon*6:
            pink = pink + 1
      if (pink == 4): ## all corners are in range
        d = box(pos=possiblePos , size=vector(l/3,l/3,l/3),
            color=vector(1,.3,.6)) # make and color
            ellipse cube
        pinkAndYellow.append(d)
        s = subCube(possiblePos , 2, possibleDist)
        (subCubes.list).append(s)
        new.append(s)
        #print "PINK"
      if count == 4:
        box(pos=possiblePos , size=vector(l/3,l/3,l/3),color
            =vector(1,0,0), opacity = .5)
        t = subCube(possiblePos , 3, possibleDist)
        (subCubes.list).append(t)
        #print "subList: ", subCubes.list[0].color
        new.append(t)
      elif possibleColor == 1:
        e= box(pos=possiblePos , size = vector(l/3,l/3,l/3)
            ,color=vector(1,1,0), opacity = .5)
        pinkAndYellow.append(e)
        u = subCube(possiblePos , 1, possibleDist)
        (subCubes.list).append(u)
        new.append(u)
      elif possibleColor == 0:
        v = subCube(possiblePos , 0, possibleDist)
        (subCubes.list).append(v)
        new.append(v)
      #else: print "SLIP"
```

```
          count = 0
          pink = 0
        c = getCoord(centerPosition)
        self.contents[c[0]][c[1]][c[2]][0]=-1
        #print "CENTER: ", getPos(c)
        for each in new:
          #print "in subdivision POS: ", each.position
          each.makeNeighbors()
          #print "in subdivision POS: ", each.position
          #print "in subdivision COLOR: ", each.color
        while new:
          del new[0]
        #print "NEW", new
      if (dimension == 3):
        for i1 in a:
          for i2 in a:
            for i3 in a:
              possiblePos = centerPosition-[i1,i2,i3]
              for j in b:
                for k in b:
                  for l in b:
                    if mag(impactPoint-possiblePos-[j,k,l]) < r
                        + epsilon: # not red and in r
                        neighborhood
                      count = count + 1
              if count == 8:
                box(pos=possiblePos, size=vector(1/3,1/3,1/3),
                    color=vector(1,0,0), opacity = .5)
                #print "SUB"
              count = 0
#Breadth-first search which labels each White cube with its
    distance to T.
#Also records best paths toward (and away from) T.
    def countFromT(self):
        if not greedy:
            return
        #label each white cube with its distance through white
             cubes to T
        currentArgs = self.contents[targetCoord[0]][
            targetCoord[1]][targetCoord[2]]
        #print "COUNT FROM T"
        currentArgs[1] = 0
        currentArgs[2] = 0
        currentArgs[3] = 0
        currentArgs[4] += 1
        iteration = int(self.arg(4,targetCoord))
        toSearch = [targetCoord]
        while len(toSearch) > 0:
            currentCoord = toSearch[0]
            del toSearch[0]
```

```
        currentArgs = self.contents[currentCoord[0]][
            currentCoord[1]][currentCoord[2]]
        currentArgs[3] = 0
        #if the current cube is Red or Pink, reset current
             args
        if currentArgs[0] > 1:
             currentArgs[1] = -1
             currentArgs[2] = 0
             currentArgs[4] = iteration
        #elif current cube should be Pink, color it pink
            and reset
        #current args
        elif mag(startPos - getPos(currentCoord)) + mag(
            targetPos-getPos(currentCoord)) > ellipseSize
            and ellipseToggle:
#if (dimension == 2):
# b = box(pos=getPos(currentCoord),size=vector(l,l,l),
    color=vector(1,.3,.6)) # ellipse color
# boxArray.append(b)
currentArgs[0] = 2
currentArgs[1] = -1
currentArgs[2] = 0
currentArgs[4] = iteration
        #else look at every neighbor for influences
        else:
             for i0 in range(-1,2):
                  for i1 in range(-1,2):
                       for i2 in range(2-dimension,dimension
                            -1):
                       #for every adjacent neighboring cube
                            if (maxCoord > currentCoord[1] +
                                i1 and maxCoord > currentCoord
                                [0] + i0 and maxCoord >
                                currentCoord[2] + i2):
             neighborCoord = [currentCoord[0]+i0,
                 currentCoord[1]+i1,currentCoord[2]+i2]
             neighborArgs = self.contents[neighborCoord
                 [0]][neighborCoord[1]][neighborCoord[2]]
             #if neighbor is Red or Pink, put in queue if
                 needed
             #but do nothing else.  Yellow cubes are
                 untouched.
             if neighborArgs[0] > 1:
               if (iteration-neighborArgs[4]>0):
                 toSearch.append(neighborCoord)
             #elif current cube is White
             elif currentArgs[0] == 0:
               #if cube hasn't been seen yet, add current
                   cube as
               #only influence
```

```
                if (iteration-neighborArgs[4]>0):
                  toSearch.append(neighborCoord)
                  neighborArgs[1] = currentArgs[1]+1
                  neighborArgs[2] = (1<<self.getIndex(
                      neighborCoord,currentCoord))
                  currentArgs[3] += (1<<self.getIndex(
                      currentCoord,neighborCoord))
                  neighborArgs[4] = iteration
                #if neighbor is labeled as too far from T,
                   add
                #current cube as only influence
                elif neighborArgs[1]-currentArgs[1]>1:
                  neighborArgs[1] = currentArgs[1]+1
                  neighborArgs[2] = (1<<self.getIndex(
                      neighborCoord,currentCoord))
                  currentArgs[3] += (1<<self.getIndex(
                      currentCoord,neighborCoord))
                #elif neighbor is influenced by current cube
                    , record
                #influence
                elif neighborArgs[1] - currentArgs[1] == 1:
                  neighborArgs[2] += (1<<self.getIndex(
                      neighborCoord,currentCoord))
                  currentArgs[3] += (1<<self.getIndex(
                      currentCoord,neighborCoord))

        if printing == True and dimension == 2:
            self.printMemory()
        for each in subCubes.list: each.greedyApp()

# Return the best choice, called D in GraphTraverse, of next
    cube to go to.
    def bestD(self,position):
        if not greedy: return self.notGreedyD(position)
        bestCoord = getCoord(position)
        lPositions = []
        neighborCoord = bestCoord
        bestPos = position
        bestDist = mag(startPos-targetPos)+1
        neigh1 = arange(bestCoord[0]-1, bestCoord[0]+2, 1)
        #for n in neigh1: print "N!: ", neigh1
        neigh2 = arange(bestCoord[1]-1, bestCoord[1]+2, 1)
        neigh3 = arange(bestCoord[2]-(dimension-2), bestCoord
            [2]+(dimension-1), 1)
        #for n2 in neigh3: print "N", neigh3
        if self.contents[bestCoord[0]][bestCoord[1]][bestCoord
            [2]][0] != -1:
      if not subD:
        #print "HERE"
```

```
  for neighborCoord in self.influences(getCoord(position
      ),2):
    if self.arg(0,neighborCoord) == 0 and mag(getPos(
        neighborCoord)-targetPos) < bestDist:
      bestPos = getPos(neighborCoord)
      bestDist = mag(getPos(neighborCoord)-targetPos)
  if mag(startPos-targetPos)+l - bestDist > 0: return
      bestPos
  else: return False
else:
  for i in neigh1:
    #print "$"
    for j in neigh2:
      #print "!"
      for k in neigh3:
        #print "H"
        if i != 0 or j != 0 or k != 0:
          neighborCoord[0] = i
          neighborCoord[1] = j
          neighborCoord[2] = k
          #print "C", neighborCoord
          if self.arg(0,neighborCoord) == 0 and mag(
              getPos(neighborCoord)-targetPos) <
              bestDist:
            bestPos = getPos(neighborCoord)
            bestDist = mag(getPos(neighborCoord)-
                targetPos)
          elif self.arg(0,neighborCoord) == -1:
            #print "NEIGH: ", getPos(neighborCoord)
            #print "CURR: ", position
            a =  getPos(neighborCoord) - position
            lPositions = dList(a, getPos(neighborCoord))
            #print "L", lPositions
            for p in lPositions:
              if mag(p-targetPos) < bestDist:
                bestDist = mag(p-targetPos)
                bestPos = p
                #print "CHOSEN"
  if mag(startPos-targetPos)+l - bestDist > 0: return
      bestPos
  else: return False
  else: # in subdivided cube
#print "P: ", position
s = subCubes.find(position)
for each in s.neighbors:
  if self.getColor(each) == -1:
    next = subCubes.find(each)
    if mag(each-targetPos) < bestDist and next.color ==
        0:
      bestPos = each
```

```
          bestDist = mag(each-targetPos)
      elif mag(each-targetPos) < bestDist and self.getColor(
          each) == 0:
        bestPos = each
        bestDist = mag(each-targetPos)
    if mag(startPos-targetPos)+l - bestDist > 0: return
        bestPos
    else: return False


  def notGreedyD(self, position):
      c = getCoord(position)
      list = []
      for i in range(-1,2):
          for j in range(-1,2):
              for k in range(2-dimension, dimension-1):
        new = [c[0]-i,c[1]-j,c[2]-k]
        if (self.contents[new[0]][new[1]][new[2]][0] == 0):
          list.append(getPos(new))
        elif self.contents[new[0]][new[1]][new[2]][0] == -1:
          sCube = subCubes.find(getPos(new))
          for s in sCube.listD(i,j,k):
            list.append(s)
      if list != []: return choice(list)
      else: return False

def resetSubDistances(self, pos):
  coord = getCoord(pos)
  best = self.contents[coord[0]][coord[1]][coord[2]][1]
  centerSubCube = subCubes.find(pos)
  if centerSubCube.tDistance +1 < best:
    best = centerSubCube.tDistance
  for subCube in centerSubCube.neighbors:
    c = subCubes.find(subCube)
    if c.tDistance +1 < best:
      best = subCube.tDistance
  self.contents[coord[0]][coord[1]][coord[2]][1] = best

  def expandEllipse(self, start):
  # reset yellow and red cubes, expand ellipse (color pink
      cubes).
  global ellipseSize
  #print "in expandEllipse: ellipseSize prior:", ellipseSize
  if not start: # no need to expand ellipse at start
    ellipseSize = ellipseSize * expand
  a = [-1.*(l/2.), (l/2.)]
  count = 0
  count2 = 0
  flag = False
```

```
for m in pinkAndYellow: # got through all pink and yellow
    cubes
    self.cubeCount += -1
   m.visible = False
   del m # delete pink or yellow cube
for i in range(maxCoord):
  for j in range(maxCoord):
    if dimension == 3:
      for k in range(maxCoord):
        if self.contents[i][j][k][0] == 0 or self.contents
            [i][j][k][0] == 2: # cube is white or pink
          for s in a:
            for t in a:
              for u in a:
                distance = mag(startPos - (getPos([i,j,k])
                    - [s,t,u])) + mag(targetPos - (getPos
                    ([i,j,k]) - [s,t,u]))
                if distance > ellipseSize and distance <=
                    ellipseSize+epsilon*6:
                  count = count + 1
                  count2 = count2 + 1
                elif distance >= ellipseSize+epsilon*6:
                  count2 = count2+1
          if count == 8 or count2 == 8:
            self.contents[i][j][k][0] = 2
            flag = True
          count = 0
          count2 = 0
        if self.contents[i][j][k][0] == 1: # is yellow
          self.contents[i][j][k][0] = 0 # color cube white
        if self.contents[i][j][k][0] == 2 and flag ==
            False: # is pink and not recently colored
          self.contents[i][j][k][0] = 0
        flag = False
    else: # 2D, so z is always 0
      if self.contents[i][j][0][0] == 0 or self.contents[i
          ][j][0][0] == 2: # white cube
        for o in a:
          for p in a: # use these to check corners
            distance = mag(startPos - (getPos([i,j,0]) - [
                o,p,0])) + mag(targetPos - (getPos([i,j
                ,0]) - [o,p,0]))
            if distance > ellipseSize and distance <=
                ellipseSize+epsilon*6:
              count = count + 1
              count2 = count2 + 1
            elif distance >= ellipseSize + epsilon * 4:
              count2= count2 + 1
        if (count == 4): ## all corners are in range
```

```
            self.contents[i][j][0][0] = 2 # set to pink cube
                 in memory
            b = box(pos=getPos([i,j,0]),size=vector(1,1,1),
                color=vector(1,.3,.6)) # make and color
                ellipse cube
            pinkAndYellow.append(b)
            self.cubeCount += 1
            flag = True
          if count2 ==4:
            self.contents[i][j][0][0] = 2
            flag = True
          count = 0
          count2 = 0
        if self.contents[i][j][0][0] == 1: #is yellow
          self.contents[i][j][0][0] = 0 # color cube white
        if self.contents[i][j][0][0] == 2 and flag == False:
            # is pink and not recently colored
          self.contents[i][j][0][0] = 0
        flag = False
    self.countFromT()

#######################################################
###                Secondary Functions              ###
#######################################################
#######################################################
#  In this section, we define:                        #
#  * the makeEnvironment function, used to populate   #
#    an Obstacles object                              #
#  * the moveTo command, which handles all movement   #
#######################################################




# Populate the environment with obstacles, and mark S and T
def makeEnvironment(obstacles):
    #for now, we'll populate the environment with random boxes
         and spheres
    decision = 0
    while not (decision == "1" or decision == "2" or decision
        == "3"):
    decision = raw_input("Press 1 to create an environment, 
        Press 2 to load an environment, Press 3 randomize the 
        course: ")
    if decision == "1":
      f = save_file()
      o = true
      t = ""
      toWrite = ""
```

```
  if f: #pass
    while (o):
      x = raw_input("Enter obstacle x coordinate: ")
      y = raw_input("Enter obstacle y coordinate: ")
      if dimension == 3:
        z = raw_input("Enter obstacle z coordinate: ")
        while (t != "s" and t != "b"): t = raw_input("
            Enter s for a sphere and b for a box: ")
      else:
        while (t != "c" and t != "b"): t = raw_input("
            Enter c for a cylinder and b for a box: ")
        z = 0
      if t == "b":
        l = raw_input("Enter obstacle length: ")
        w = raw_input("Enter obstacle width: ")
        h = raw_input("Enter obstacle height: ")
        new = box(pos=[int(x),int(y),int(z)], size=(int(l)
            ,int(w),int(h)), color=(0,.5,1), opacity = .5)
      else:
        r = raw_input("Enter obstacle radius: ")
        if t == "c": new = cylinder(pos=[int(x),int(y),int
            (z)],axis=(0,0,9),radius= int(r), opacity =
            .5, color = (0,.3,1))
        else: new = sphere(pos=[int(x),int(y),int(z)],
            radius = int(r), color = (0,.3,1), opacity =
            .5 )
      obstacles.add(new)
      if t == "b": toWrite = str(x)+" " + str(y)+" " + str
          (z)+" " + str(t)+" " + str(l) + " " + str(w)+" "
           + str(h)+ "\n"
      else: toWrite = str(x)+" " + str(y)+" " + str(z)+" "
           + str(t)+" " + str(r) + "\n"
      continuation = raw_input("Would you like to add 
          another obstacle (1 = yes, anything else = no): 
          ")
      f.write(toWrite)
      if continuation == "1":
        o = true
        t = ""
      else:
        o = false
        f.close()
elif decision == "2":
  f = get_file()
  if f:
    data = f.readlines()
    f.close()
    for line in data:
      list = line.split()
      x = list[0]
```

```
          y = list[1]
          z = list[2]
          type = list[3]
          if type == "b": new = box(pos=[float(x),float(y),
              float(z)], size=(float(list[4]),float(list[5]),
              float(list[6])), color=(0,.5,1), opacity = .5)
          elif type == "c": new = cylinder(pos=[float(x),float
              (y),float(z)],axis=(0,0,9),radius= float(list
              [4]), opacity = .5, color = (0,.3,1))
          else: new = sphere(pos=[float(x),float(y),float(z)],
               radius = float(list[4]), color = (0,.3,1),
              opacity = .5 )
          obstacles.add(new)
    elif decision == "3":
      f = save_file()
      if f:
        toWrite = ""
        if dimension == 2:
          for i in range(0,15):
            toWrite = obstacles.addRandomBox()
            f.write(toWrite)
            toWrite = obstacles.addRandomCylinder()
            f.write(toWrite)
        if dimension == 3:
          for i in range(0,15):
            toWrite = obstacles.addRandomBox()
            f.write(toWrite)
            toWrite = obstacles.addRandomSphere()
            f.write(toWrite)
        f.close()


# color all cubes red near any intersection points between the
     obstacles
# and Bob's current position
def colorCubesRed(next):
  for thing in obstacles.contents:
    if thing.intersects(Bob):
      if (type(thing) == sphereType):
        intersectionPoint = thing.pos + (thing.radius/mag(Bob.
            pos-thing.pos))*(Bob.pos-thing.pos)
      elif (type(thing) == cylinderType):
        thingpos = vector(thing.pos)
        thingpos[2] = 0
        intersectionPoint = thingpos + (thing.radius/mag(Bob.
            pos-thingpos))*(Bob.pos-thingpos)
      elif (type(thing) == boxType):
        cornerlo = thing.pos - 0.5*thing.size
        cornerhi = thing.pos + 0.5*thing.size
        center = vector(Bob.pos)
```

```
        intersectionPoint = vector(Bob.pos)
        for i in [0,1,2]:
          if   center[i] < cornerlo[i]:
            intersectionPoint[i] = cornerlo[i]
          elif center[i] > cornerhi[i]:
            intersectionPoint[i] = cornerhi[i]
      if (maxColor):
        return cubes.addObstacle(intersectionPoint,next)
      else:
        cubes.recolor(getCoord(next),3)
        if (cubes.notifyNeighbors(getCoord(next),
            intersectionPoint)):
          cubes.countFromT()
        return [0]


def moveTo(nextPos):
  global moves
  global dMove
  global subMove
  global dSubMove
  global totalDistance
  #print "MoveTO"
  #print "nextPOS: ", nextPos
  h = getCoord(nextPos)
  subFlag = False
  sFlag = False
  moveC = True
  totalDistance = totalDistance + mag(Bob.pos - nextPos)
  if cubes.contents[h[0]][h[1]][h[2]][0] == -1:
    subMove = subMove + 1
    subFlag = True
    if ( mag(nextPos - Bob.pos) > (l*(2.**(1./2.)))/3   ):
      sFlag = True
      dSubMove = dSubMove + 1
  else: moves = moves + 1
  flag = False
  #print l
  #print l*(2.**(1./2.))
  #print mag(nextPos - Bob.pos)
  if ( mag(nextPos - Bob.pos) >= l*(2.**(1./2.))   ):
    dMove = dMove + 1
    #totalDistance = totalDistance + 2**(1/2.)*l
    flag = True
  if printing == True:
    print getCoord(nextPos), cubes.arg(1,getCoord(nextPos)),
        binaryString(cubes.arg(2,getCoord(nextPos)))
  originalPos = vector(Bob.pos)
  #print "MAG1 ", mag(Bob.pos-nextPos)
  while mag(Bob.pos-nextPos)>epsilon/20:
```

```
  #print mag(Bob.pos-nextPos)
  #print epsilon/20
  rate(100)
  Bob.pos += .1*normalize(nextPos-Bob.pos)
  #print Bob.pos
  scene.center = Bob.pos
      if obstacles.intersects(Bob):        #if an obstacle is
           encountered,
          if printing == True:
              print "...OBSTACLE.  Back to ", getCoord(
                  originalPos), binaryString(cubes.arg(2,
                  getCoord(originalPos)))
          returnValue = colorCubesRed(nextPos)
          #print "r"
          #print returnValue
          if printing == True:
              print "(influences after recolor: ",
                  binaryString(cubes.arg(2,getCoord(
                  originalPos))), ")"
          #print "3"
          if (flag): dMove = dMove + 1
          if subFlag: subMove = subMove + 1
          else: moves = moves + 1
          if sFlag: dSubMove = dSubMove + 1
          #print "move2 ", mag(Bob.pos-originalPos)
          while mag(Bob.pos-originalPos)>epsilon/20.:    #
              return to original position # originally
              epsilon/2
              #print "4"
              rate(100)
              Bob.pos += .1*normalize(originalPos-Bob.pos)
              scene.center = Bob.pos
          #for nothing in range(10):
    # rate(10)
    # Bob.pos = 1/10.*normalize(originalPos - Bob.pos)
          Bob.pos = originalPos
          if moveC:
      totalDistance = totalDistance + mag(originalPos -
          nextPos)
      moveC = False
          return returnValue                #return that
              moveTo failed
if (cubes.getColor(nextPos) != 1 and cubes.getColor(nextPos)
     != -1): cubes.recolor(getCoord(nextPos),1)        #next
    cube successfully reached;
elif (cubes.getColor(nextPos) == -1):
  newS = subCubes.find(nextPos)
  newS.recolor(1)
Bob.pos = nextPos
```

```
  return [1]                          #color next Yellow,
      return True

###################################################
###             Boxes and GraphTraverse          ###
###################################################
cubes = CubeArray()
subCubes = SubList()
#subList = SubCubeList()
#subList = []
global ellipseSize
ellipseSize = (1.1*mag(startPos-targetPos))

def Boxes():
  global ellipseSize
  cubes.countFromT()
  test = True
  if ellipseToggle: cubes.expandEllipse(True)
  moveReturned = moveTo(getPos(startCoord))
  #print moveReturned
  #print (moveReturned[0] != 1)
  #print moveReturned[0]
  if moveReturned[0] != 1: return False
  while(test):
    result = GraphTraverse()
    #print result
    if (result == [1]):
      test = False
    elif (result == [2]):
      return False
    elif ellipseToggle: cubes.expandEllipse(False)
    else: return False
  moveReturned = moveTo(targetPos)
  if moveReturned[0] != 1: return False
  else: return True


def GraphTraverse():
    if printing == True:
        print "Cube count:  ", cubes.cubeCount
    thisPos = vector(Bob.pos)
    if mag(vector(targetCoord)-vector(getCoord(Bob.pos)))<1: #
         here is the check for target
        return [1]
    D = cubes.bestD(Bob.pos)
    while D:
        moveReturned = moveTo(D)
        #print "D: ", D
        #print moveReturned
        if moveReturned[0] == 1:
```

```
        GTReturned = GraphTraverse()
        #print "GT: ", GTReturned
        if GTReturned[0] == -1:
            moveTo(thisPos)
            #print "GT1: ", mag(Bob.pos - getPos(
                GTReturned[1]))
            #print "ep/2: ", epsilon/2.
            if mag(Bob.pos - getPos(GTReturned[1]))>
                epsilon/2.:
      return GTReturned
            else:
                return [1]
        if GTReturned[0] == 0:
            return GTReturned

        if cubes.arg(0,targetCoord) == 3: #if target cube
            is Red, stop
            return [2]
        if GTReturned[0] == 2:
            return GTReturned
        if GTReturned[0] == 4:
    moveTo(thisPos)
    cubes.countFromT()
    if cubes.bestD(thisPos): return GraphTraverse()
    else: return [4]
        #If you still have neighbors, make sure you're in
            the right spot
        if GTReturned[0] == 3:
            if cubes.bestD(thisPos):
                while GTReturned[1] != []:
                    if printing:  print "...Back to ",
                        getCoord(GTReturned[1][0])
                    moveTo(GTReturned[1][0])
                    del GTReturned[1][0]
            else:
      #print "GT3"
      GTReturned[1].append(thisPos)
      return GTReturned

    elif moveReturned[0] == -1:  #if a Yellow cube has
        been colored Red:
        moveTo(thisPos)
        #print "GT2: ", mag(Bob.pos - getPos(moveReturned
            [1]))
        #print "ep/2: ", epsilon/2.
        #if mag(Bob.pos - getPos(moveReturned[1]))>epsilon
            /2.:
        return [4]#moveReturned
#else:
        #return [1]
```

```
        if mag(vector(targetCoord)-vector(getCoord(Bob.pos)))
            <1:# here is the check for target
            return [1]
        D = cubes.bestD(Bob.pos)
    return [3,[thisPos]]


########################################################
###                    Main Function                 ###
########################################################

obstacles = Obstacles()
#print "H"
makeEnvironment(obstacles)
#print epsilon
#print l

if(Boxes()):
  print "TARGET REACHED"
else:
  print "TARGET UNREACHABLE"
#if dimension == 2:
  #cubes.printMemory()
#print "total number of cubes: ", cubes.cubeCount
#print "total number of moves: ", moves
#print "total number of diagonal moves: ", dMove
#print "total number of sudivisional moves: ", subMove
#print "total number of diagonal subdivisional moves: ",
    dSubMove
#print "l: ", l
print "total Distance: ", totalDistance
```