Navigating Blindly

Multidimensional online robot motion in unknown environments

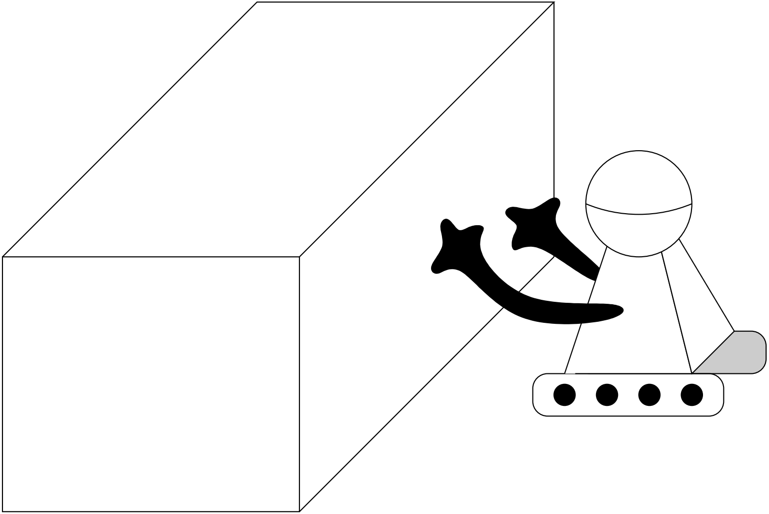
Josh Brown Kramer
Illinois Wesleyan University

Lucas Sabalka
University of California, Davis,
soon to be at Binghamton University

22 October, 2008

Outline

Navigate Blindly



Our Three Tasks

- Given:
  - An environment $X \subset \mathbb{R}^n$ with finite diameter
  - A spherical robot, named Bob, of radius $r > 0$, equipped with:
    - tactile sensor
    - GPS sensor
  - A starting point $S \in X$
  - Possibly, a target point $T \in X$
- The tasks are:
  - $COVER_n$: Occupy as much of X as possible
  - $SEARCH_n$: Find T and move from S to T
  - NAV_n: Move from S to T
- We want an *efficient* algorithm solving the task.
  - offline if X is known
  - online if X is unknown

Our Three Tasks

- Given:
  - An environment $X \subset \mathbb{R}^n$ with finite diameter
  - A spherical robot, named Bob, of radius $r > 0$, equipped with:
    - tactile sensor
    - GPS sensor
  - A starting point $S \in X$
  - Possibly, a target point $T \in X$
- The tasks are:
  - $COVER_n$: Occupy as much of X as possible
  - $SEARCH_n$: Find T and move from S to T
  - NAV_n: Move from S to T
- We want an *efficient* algorithm solving the task.
  - offline if X is known
  - online if X is unknown

Our Three Tasks

- Given:
  - An environment $X \subset \mathbb{R}^n$ with finite diameter
  - A spherical robot, named Bob, of radius $r > 0$, equipped with:
    - tactile sensor
    - GPS sensor
  - A starting point $S \in X$
  - Possibly, a target point $T \in X$
- The tasks are:
  - $COVER_n$: Occupy as much of X as possible
  - $SEARCH_n$: Find T and move from S to T
  - NAV_n: Move from S to T
- We want an *efficient* algorithm solving the task.
  - offline if X is known
  - online if X is unknown

Our Three Tasks

- Given:
  - An environment $X \subset \mathbb{R}^n$ with finite diameter
  - A spherical robot, named Bob, of radius $r > 0$, equipped with:
    - tactile sensor
    - GPS sensor
  - A starting point $S \in X$
  - Possibly, a target point $T \in X$
- The tasks are:
  - $COVER_n$: Occupy as much of X as possible
  - $SEARCH_n$: Find T and move from S to T
  - NAV_n: Move from S to T
- We want an *efficient* algorithm solving the task.
  - offline if X is known
  - online if X is unknown

Our Three Tasks

- Given:
  - An environment $X \subset \mathbb{R}^n$ with finite diameter
  - A spherical robot, named Bob, of radius $r > 0$, equipped with:
    - tactile sensor
    - GPS sensor
  - A starting point $S \in X$
  - Possibly, a target point $T \in X$
- The tasks are:
  - $COVER_n$: Occupy as much of X as possible
  - $SEARCH_n$: Find T and move from S to T
  - NAV_n: Move from S to T
- We want an *efficient* algorithm solving the task.
  - offline if X is known
  - online if X is unknown

Our Three Tasks

- Given:
  - An environment $X \subset \mathbb{R}^n$ with finite diameter
  - A spherical robot, named Bob, of radius $r > 0$, equipped with:
    - tactile sensor
    - GPS sensor
  - A starting point $S \in X$
  - Possibly, a target point $T \in X$
- The tasks are:
  - $COVER_n$: Occupy as much of X as possible
  - $SEARCH_n$: Find T and move from S to T
  - NAV_n: Move from S to T
- We want an *efficient* algorithm solving the task.
  - offline if X is known
  - online if X is unknown

Our Three Tasks

- Given:
  - An environment $X \subset \mathbb{R}^n$ with finite diameter
  - A spherical robot, named Bob, of radius $r > 0$, equipped with:
    - tactile sensor
    - GPS sensor
  - A starting point $S \in X$
  - Possibly, a target point $T \in X$
- The tasks are:
  - $COVER_n$: Occupy as much of X as possible
  - $SEARCH_n$: Find T and move from S to T
  - NAV_n: Move from S to T
- We want an *efficient* algorithm solving the task.
  - offline if X is known
  - online if X is unknown

Our Three Tasks

- Given:
  - An environment $X \subset \mathbb{R}^n$ with finite diameter
  - A spherical robot, named Bob, of radius $r > 0$, equipped with:
    - tactile sensor
    - GPS sensor
  - A starting point $S \in X$
  - Possibly, a target point $T \in X$
- The tasks are:
  - $COVER_n$: Occupy as much of X as possible
  - $SEARCH_n$: Find T and move from S to T
  - NAV_n: Move from S to T
- We want an *efficient* algorithm solving the task.
  - offline if X is known
  - online if X is unknown

Our Three Tasks

- Given:
  - An environment $X \subset \mathbb{R}^n$ with finite diameter
  - A spherical robot, named Bob, of radius $r > 0$, equipped with:
    - tactile sensor
    - GPS sensor
  - A starting point $S \in X$
  - Possibly, a target point $T \in X$
- The tasks are:
  - $COVER_n$: Occupy as much of X as possible
  - $SEARCH_n$: Find T and move from S to T
  - NAV_n: Move from S to T
- We want an *efficient* algorithm solving the task.
  - offline if X is known
  - online if X is unknown

Our Three Tasks

- Given:
  - An environment $X \subset \mathbb{R}^n$ with finite diameter
  - A spherical robot, named Bob, of radius $r > 0$, equipped with:
    - tactile sensor
    - GPS sensor
  - A starting point $S \in X$
  - Possibly, a target point $T \in X$
- The tasks are:
  - $COVER_n$: Occupy as much of X as possible
  - $SEARCH_n$: Find T and move from S to T
  - NAV_n: Move from S to T
- We want an *efficient* algorithm solving the task.
  - offline if X is known
  - online if X is unknown

Our Three Tasks

- Given:
  - An environment $X \subset \mathbb{R}^n$ with finite diameter
  - A spherical robot, named Bob, of radius $r > 0$, equipped with:
    - tactile sensor
    - GPS sensor
  - A starting point $S \in X$
  - Possibly, a target point $T \in X$
- The tasks are:
  - $COVER_n$: Occupy as much of X as possible
  - $SEARCH_n$: Find T and move from S to T
  - NAV_n: Move from S to T
- We want an *efficient* algorithm solving the task.
  - offline if X is known
  - online if X is unknown

Our Three Tasks

- Given:
  - An environment $X \subset \mathbb{R}^n$ with finite diameter
  - A spherical robot, named Bob, of radius $r > 0$, equipped with:
    - tactile sensor
    - GPS sensor
  - A starting point $S \in X$
  - Possibly, a target point $T \in X$
- The tasks are:
  - $COVER_n$: Occupy as much of X as possible
  - $SEARCH_n$: Find T and move from S to T
  - NAV_n: Move from S to T
- We want an *efficient* algorithm solving the task.
  - offline if X is known
  - online if X is unknown

Why?

- navigation problems
  - mail delivery in a city
  - moving packages in a factory
- configuration space problems
  - Shuttle arm motion
- exploration and sample acquisition
  - Mars Rover
- area coverage problems
  - cleaning public places
  - Roomba (video)

Why?

- navigation problems
    - mail delivery in a city
    - moving packages in a factory
- configuration space problems
    - Shuttle arm motion
- exploration and sample acquisition
    - Mars Rover
- area coverage problems
    - cleaning public places
    - Roomba (video)

Why?

- navigation problems
  - mail delivery in a city
  - moving packages in a factory
- configuration space problems
  - Shuttle arm motion
- exploration and sample acquisition
  - Mars Rover
- area coverage problems
  - cleaning public places
  - Roomba (video)

Why?

- navigation problems
  - mail delivery in a city
  - moving packages in a factory
- configuration space problems
  - Shuttle arm motion
- exploration and sample acquisition
  - Mars Rover
- area coverage problems
  - cleaning public places
  - Roomba (video)

Why?

- navigation problems
  - mail delivery in a city
  - moving packages in a factory
- configuration space problems
  - Shuttle arm motion
- exploration and sample acquisition
  - Mars Rover
- area coverage problems
  - cleaning public places
  - Roomba (video)

Why?

- navigation problems
  - mail delivery in a city
  - moving packages in a factory
- configuration space problems
  - Shuttle arm motion
- exploration and sample acquisition
  - Mars Rover
- area coverage problems
  - cleaning public places
  - Roomba (video)

Why?

- navigation problems
  - mail delivery in a city
  - moving packages in a factory
- configuration space problems
  - Shuttle arm motion
- exploration and sample acquisition
  - Mars Rover
- area coverage problems
  - cleaning public places
  - Roomba (video)

Why?

- navigation problems
  - mail delivery in a city
  - moving packages in a factory
- configuration space problems
  - Shuttle arm motion
- exploration and sample acquisition
  - Mars Rover
- area coverage problems
  - cleaning public places
  - Roomba (video)

Why?

- navigation problems
  - mail delivery in a city
  - moving packages in a factory
- configuration space problems
  - Shuttle arm motion
- exploration and sample acquisition
  - Mars Rover
- area coverage problems
  - cleaning public places
  - Roomba (video)

Why?

- navigation problems
  - mail delivery in a city
  - moving packages in a factory
- configuration space problems
  - Shuttle arm motion
- exploration and sample acquisition
  - Mars Rover
- area coverage problems
  - cleaning public places
  - Roomba (video)

Mathematical Motivation

- Discrete approximation via Rips complexes

Theorem (Caraballo)

Let C be a compact subset of $\mathbb{R}^n$. For any point $q \in \mathbb{R}^n$ and for almost every $r > 0$:

$$Vol_{n-1}((d_C^{-1}(r)) \cap B^n(q, 2r)) \leq 4^{n+1} r^{n-1}.$$

- Even if C is a fractal curve, most tubes about C have finite surface area.
- Let C be the set of points where Bob's center can be. Almost always, the boundary of C has finite volume.

Mathematical Motivation

- Discrete approximation via Rips complexes

Theorem (Caraballo)

Let C be a compact subset of $\mathbb{R}^n$. For any point $q \in \mathbb{R}^n$ and for almost every $r > 0$:

$$Vol_{n-1}((d_C^{-1}(r)) \cap B^n(q, 2r)) \leq 4^{n+1} r^{n-1}.$$

- Even if C is a fractal curve, most tubes about C have finite surface area.
- Let C be the set of points where Bob's center can be. Almost always, the boundary of C has finite volume.

Mathematical Motivation

- Discrete approximation via Rips complexes

Theorem (Caraballo)

Let C be a compact subset of $\mathbb{R}^n$. For any point $q \in \mathbb{R}^n$ and for almost every $r > 0$:

$$Vol_{n-1}((d_C^{-1}(r)) \cap B^n(q, 2r)) \leq 4^{n+1} r^{n-1}.$$

- Even if C is a fractal curve, most tubes about C have finite surface area.
- Let C be the set of points where Bob's center can be. Almost always, the boundary of C has finite volume.

Mathematical Motivation

- Discrete approximation via Rips complexes

Theorem (Caraballo)

Let C be a compact subset of $\mathbb{R}^n$. For any point $q \in \mathbb{R}^n$ and for almost every $r > 0$:

$$Vol_{n-1}((d_C^{-1}(r)) \cap B^n(q, 2r)) \leq 4^{n+1} r^{n-1}.$$

- Even if C is a fractal curve, most tubes about C have finite surface area.
- Let C be the set of points where Bob's center can be. Almost always, the boundary of C has finite volume.

The BUG1 algorithm

[Lumelsky and Stepanov]

BUG1

While not at T:

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

Target reached

The BUG1 algorithm

[Lumelsky and Stepanov]

BUG1

While not at T:

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

Target reached

The BUG1 algorithm

[Lumelsky and Stepanov]

BUG1

While not at T:

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

Target reached

The BUG1 algorithm

[Lumelsky and Stepanov]

BUG1

While not at T:

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

Target reached

The BUG1 algorithm

[Lumelsky and Stepanov]

BUG1

While not at T:

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

Target reached

The BUG1 algorithm

[Lumelsky and Stepanov]

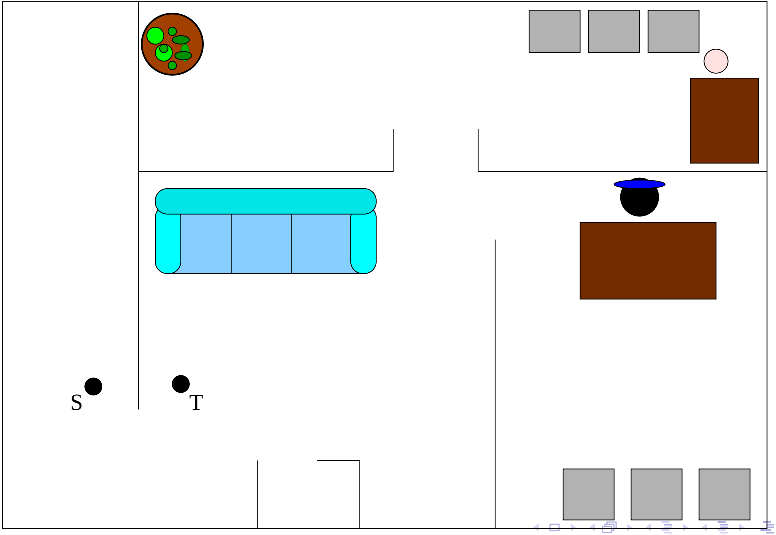
BUG1

While not at T:

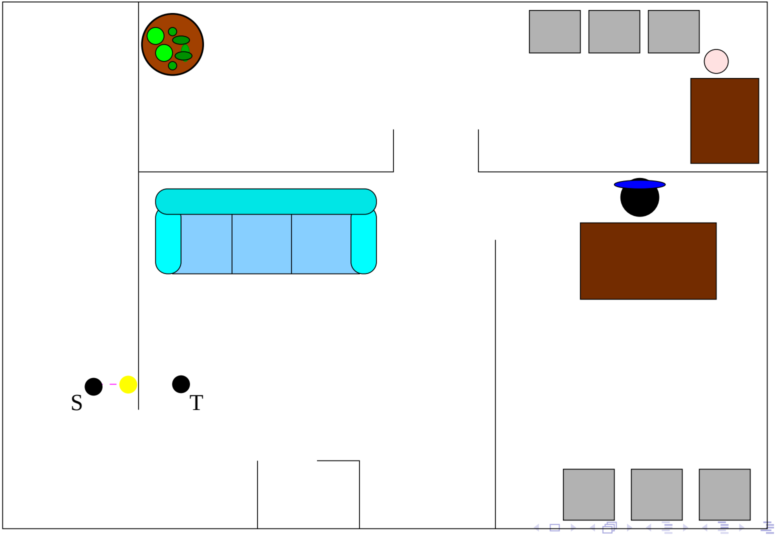
- Move directly towards T.
- **If** an obstacle is encountered:
  - Explore the obstacle (via clockwise circumnavigation).
  - *Move* to some point p_{min} on the obstacle closest to T.
  - **If** Bob cannot move directly towards T from p_{min}:
    - Target unreachable.

Target reached

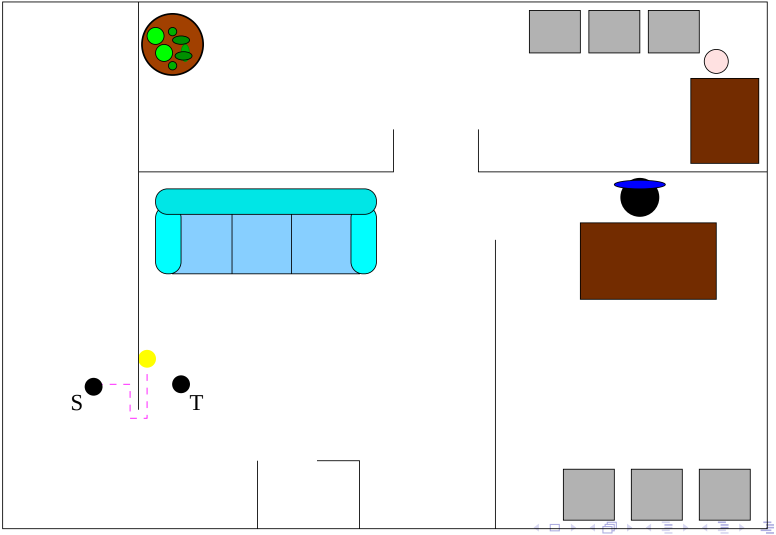
BUG1 Example



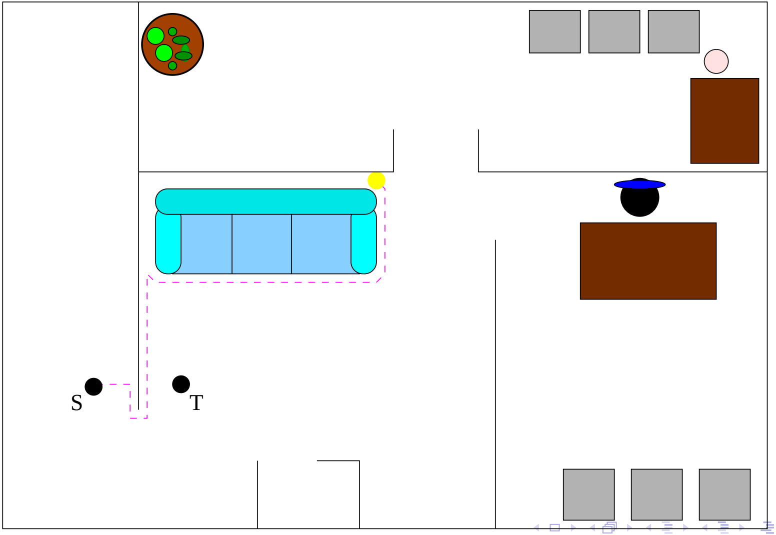
BUG1 Example



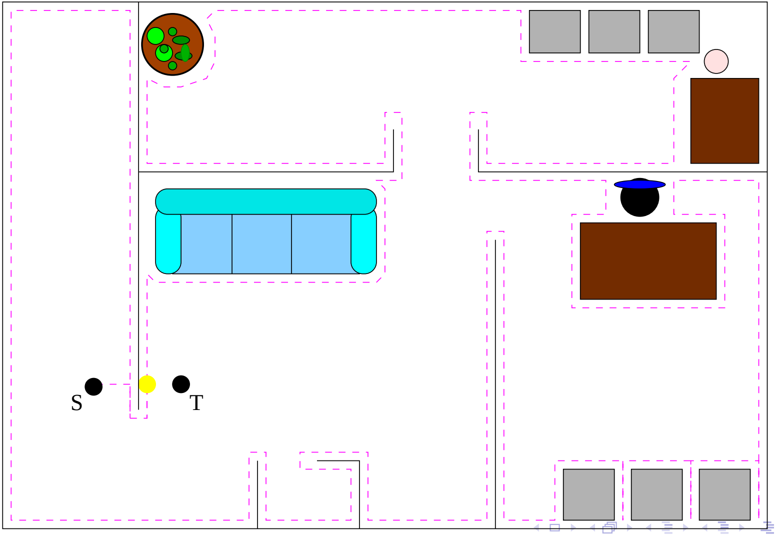
BUG1 Example



BUG1 Example



BUG1 Example



Definition of Competitiveness

- Given task P, like NAV_n
- For any algorithm A solving P, define

$$f_A(t) = \sup\{t_A(X) | t_{opt}(X) \leq t\}$$

- $g : \mathbb{R} \to \mathbb{R}$ is a *universal lower bound* on competitiveness of P if for all A,

$$f_A \in \Omega(g)$$

- A is $O(g)$*-competitive* if

$$f_A \in O(g)$$

- A is *optimally competitive* if A is $O(g)$-competitive and g is a ULB for P

Definition of Competitiveness

- Given task P, like NAV_n
- For any algorithm A solving P, define

$$f_A(t) = \sup\{t_A(X) | t_{opt}(X) \leq t\}$$

- $g : \mathbb{R} \to \mathbb{R}$ is a *universal lower bound* on competitiveness of P if for all A,

$$f_A \in \Omega(g)$$

- A is *$O(g)$-competitive* if

$$f_A \in O(g)$$

- A is *optimally competitive* if A is $O(g)$-competitive and g is a ULB for P

Definition of Competitiveness

- Given task P, like NAV_n
- For any algorithm A solving P, define

$$f_A(t) = \sup\{t_A(X) | t_{opt}(X) \leq t\}$$

- $g : \mathbb{R} \to \mathbb{R}$ is a *universal lower bound* on competitiveness of P if for all A,

$$f_A \in \Omega(g)$$

- A is *$O(g)$-competitive* if

$$f_A \in O(g)$$

- A is *optimally competitive* if A is $O(g)$-competitive and g is a ULB for P

Definition of Competitiveness

- Given task P, like NAV_n
- For any algorithm A solving P, define

$$f_A(t) = \sup\{t_A(X) | t_{opt}(X) \leq t\}$$

- $g : \mathbb{R} \to \mathbb{R}$ is a *universal lower bound* on competitiveness of P if for all A,

$$f_A \in \Omega(g)$$

- A is $O(g)$*-competitive* if

$$f_A \in O(g)$$

- A is *optimally competitive* if A is $O(g)$-competitive and g is a ULB for P

Definition of Competitiveness

- Given task P, like NAV_n
- For any algorithm A solving P, define

$$f_A(t) = \sup\{t_A(X) | t_{opt}(X) \leq t\}$$

- $g : \mathbb{R} \to \mathbb{R}$ is a *universal lower bound* on competitiveness of P if for all A,

$$f_A \in \Omega(g)$$

- A is *$O(g)$-competitive* if

$$f_A \in O(g)$$

- A is *optimally competitive* if A is $O(g)$-competitive and g is a ULB for P

Competitiveness

- Competitiveness can be linear, quadratic, exponential, etc.
- A is linearly competitive iff $f_A(t) \in O(t)$ iff there exists c_1, c_0 with $t_A(t) \leq c_1 t_{opt} + c_0$
- Example: Tree traversal.
  - Goal: visit each vertex of a tree and return to start.
  - Algorithm: Never go back across an edge until all neighboring edges have been traversed twice.

Competitiveness

- Competitiveness can be linear, quadratic, exponential, etc.
- A is linearly competitive iff $f_A(t) \in O(t)$ iff there exists c_1, c_0 with $t_A(t) \leq c_1 t_{opt} + c_0$
- Example: Tree traversal.
  - Goal: visit each vertex of a tree and return to start.
  - Algorithm: Never go back across an edge until all neighboring edges have been traversed twice.

Competitiveness

- Competitiveness can be linear, quadratic, exponential, etc.
- A is linearly competitive iff $f_A(t) \in O(t)$ iff there exists c_1, c_0 with $t_A(t) \leq c_1 t_{opt} + c_0$
- Example: Tree traversal.
  - Goal: visit each vertex of a tree and return to start.
  - Algorithm: Never go back across an edge until all neighboring edges have been traversed twice.

Competitiveness

- Competitiveness can be linear, quadratic, exponential, etc.
- A is linearly competitive iff $f_A(t) \in O(t)$ iff there exists c_1, c_0 with $t_A(t) \leq c_1 t_{opt} + c_0$
- Example: Tree traversal.
  - Goal: visit each vertex of a tree and return to start.
  - Algorithm: Never go back across an edge until all neighboring edges have been traversed twice.

Competitiveness

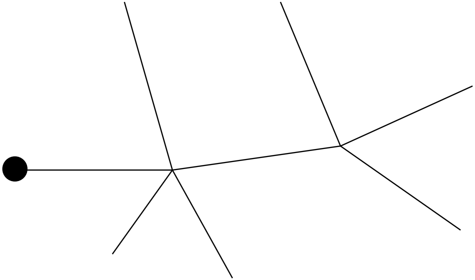
- Competitiveness can be linear, quadratic, exponential, etc.
- A is linearly competitive iff $f_A(t) \in O(t)$ iff there exists c_1, c_0 with $t_A(t) \leq c_1 t_{opt} + c_0$
- Example: Tree traversal.
  - Goal: visit each vertex of a tree and return to start.
  - Algorithm: Never go back across an edge until all neighboring edges have been traversed twice.

Competitiveness

- Competitiveness can be linear, quadratic, exponential, etc.
- A is linearly competitive iff $f_A(t) \in O(t)$ iff there exists c_1, c_0 with $t_A(t) \leq c_1 t_{opt} + c_0$
- Example: Tree traversal.
  - Goal: visit each vertex of a tree and return to start.
  - Algorithm: Never go back across an edge until all neighboring edges have been traversed twice.

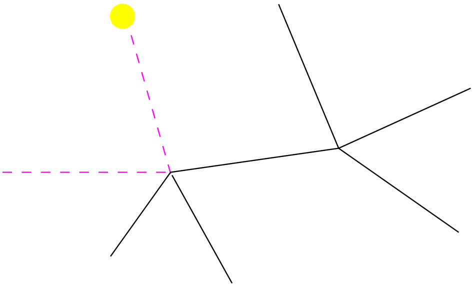
Competitiveness

- Competitiveness can be linear, quadratic, exponential, etc.
- A is linearly competitive iff $f_A(t) \in O(t)$ iff there exists c_1, c_0 with $t_A(t) \leq c_1 t_{opt} + c_0$
- Example: Tree traversal.
  - Goal: visit each vertex of a tree and return to start.
  - Algorithm: Never go back across an edge until all neighboring edges have been traversed twice.



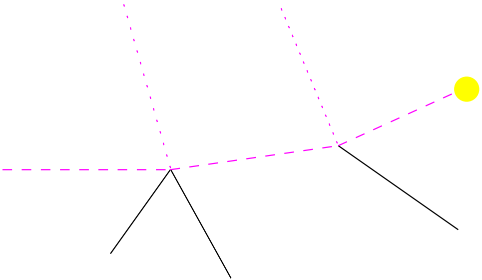
Competitiveness

- Competitiveness can be linear, quadratic, exponential, etc.
- A is linearly competitive iff $f_A(t) \in O(t)$ iff there exists c_1, c_0 with $t_A(t) \leq c_1 t_{opt} + c_0$
- Example: Tree traversal.
  - Goal: visit each vertex of a tree and return to start.
  - Algorithm: Never go back across an edge until all neighboring edges have been traversed twice.



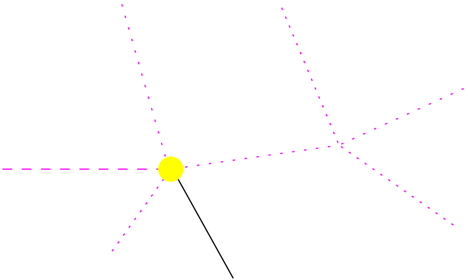
Competitiveness

- Competitiveness can be linear, quadratic, exponential, etc.
- A is linearly competitive iff $f_A(t) \in O(t)$ iff there exists c_1, c_0 with $t_A(t) \leq c_1 t_{opt} + c_0$
- Example: Tree traversal.
  - Goal: visit each vertex of a tree and return to start.
  - Algorithm: Never go back across an edge until all neighboring edges have been traversed twice.



Competitiveness

- Competitiveness can be linear, quadratic, exponential, etc.
- A is linearly competitive iff $f_A(t) \in O(t)$ iff there exists c_1, c_0 with $t_A(t) \leq c_1 t_{opt} + c_0$
- Example: Tree traversal.
  - Goal: visit each vertex of a tree and return to start.
  - Algorithm: Never go back across an edge until all neighboring edges have been traversed twice.



Competitiveness

- Competitiveness can be linear, quadratic, exponential, etc.
- A is linearly competitive iff $f_A(t) \in O(t)$ iff there exists c_1, c_0 with $t_A(t) \leq c_1 t_{opt} + c_0$
- Example: Tree traversal.
  - Goal: visit each vertex of a tree and return to start.
  - Algorithm: Never go back across an edge until all neighboring edges have been traversed twice.
  - Must traverse each edge twice, and algorithm traverses each edge exactly twice, so optimally (linearly) competitive.

Competitiveness of BUG1

Competitiveness of BUG1?

- Horrible.
- Runs in time proportional to sum of lengths of boundaries of (intervening) obstacles.
- Is not $O(g)$-competitive for any function g.

How to fix it? Don't allow Bob to get too far from T. Introduce ellipse as virtual obstacle.

Competitiveness of BUG1

Competitiveness of BUG1?

- Horrible.
- Runs in time proportional to sum of lengths of boundaries of (intervening) obstacles.
- Is not $O(g)$-competitive for any function g.

How to fix it? Don't allow Bob to get too far from T. Introduce ellipse as virtual obstacle.

Competitiveness of BUG1

Competitiveness of BUG1?

- Horrible.
- Runs in time proportional to sum of lengths of boundaries of (intervening) obstacles.
- Is not $O(g)$-competitive for any function g.

How to fix it? Don't allow Bob to get too far from T. Introduce ellipse as virtual obstacle.

Competitiveness of BUG1

Competitiveness of BUG1?

- Horrible.
- Runs in time proportional to sum of lengths of boundaries of (intervening) obstacles.
- Is not $O(g)$-competitive for any function g.

How to fix it? Don't allow Bob to get too far from T. Introduce ellipse as virtual obstacle.

CBUG

[Gabriely and Rimon]

CBUG

Fix initial area $A_0(\sim d(S,T)^2)$

For $i = 0$ to ∞:

- **Execute** BUG1(S,T) within ellipse with foci S and T and area $2^i A_0$.
- Success if at T
- Failure if Bob did not touch virtual ellipse

CBUG

[Gabriely and Rimon]

CBUG

Fix initial area $A_0(\sim d(S,T)^2)$

For $i = 0$ to ∞:

- **Execute** BUG1(S,T) within ellipse with foci S and T and area $2^i A_0$.
- Success if at T
- Failure if Bob did not touch virtual ellipse

CBUG

[Gabriely and Rimon]

CBUG

Fix initial area $A_0 (\sim d(S, T)^2)$

For $i = 0$ to ∞:

- **Execute** BUG1(S, T) within ellipse with foci S and T and area $2^i A_0$.
- Success if at T
- Failure if Bob did not touch virtual ellipse

CBUG

[Gabriely and Rimon]

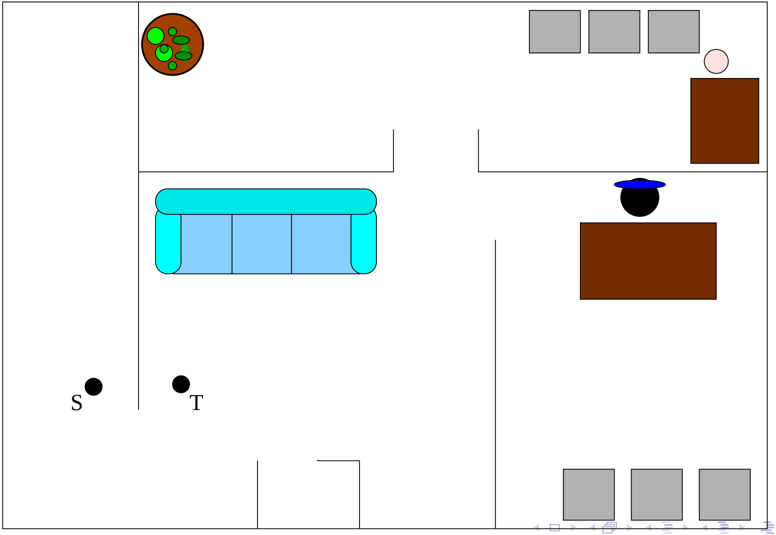
CBUG

Fix initial area $A_0 (\sim d(S,T)^2)$

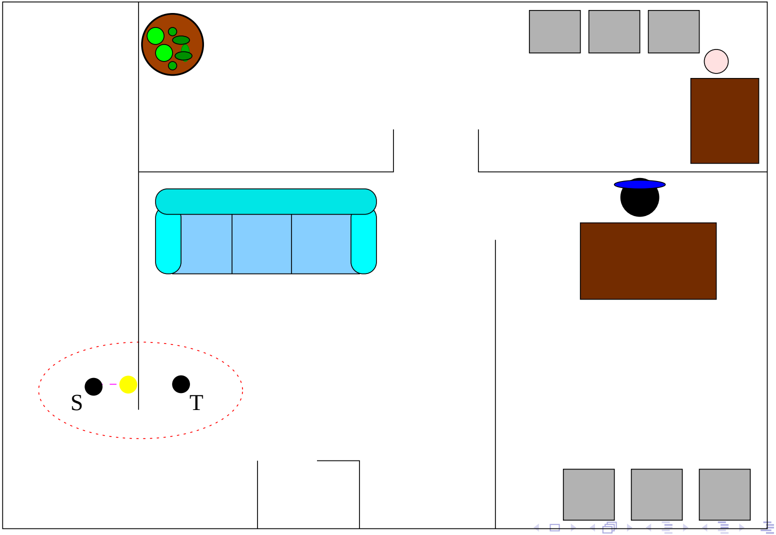
For $i = 0$ to ∞:

- **Execute** BUG1(S,T) within ellipse with foci S and T and area $2^i A_0$.
- Success if at T
- Failure if Bob did not touch virtual ellipse

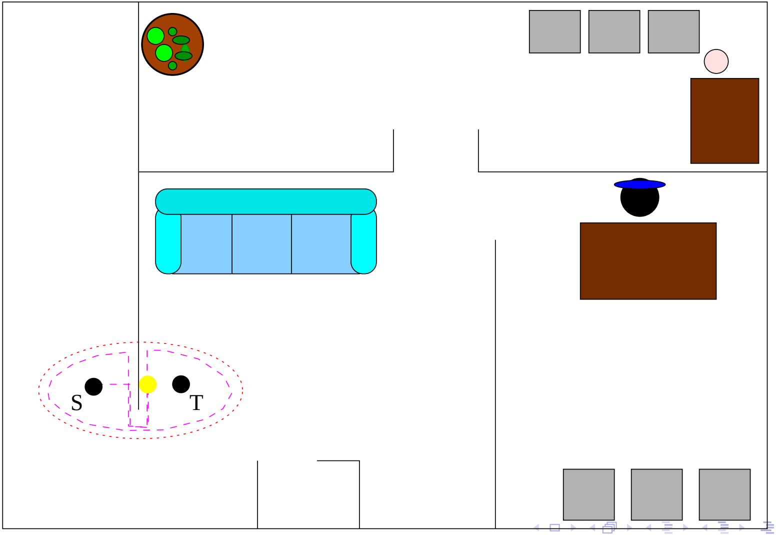
CBUG Example



CBUG Example



CBUG Example



Competitiveness of CBUG

(Show Program)

Geometric Progression.

Theorem (Gabriely and Rimon)

NAV_2 has a quadratic universal lower bound, namely given by

$$g_r(x) := \frac{2\pi}{3(1+\pi)^2 r} x^2 \sim \frac{.122x^2}{r}.$$

Theorem (Gabriely and Rimon)

If T is reachable, CBUG solves NAV_2 in time at most

$$\frac{3\pi}{r} l_{opt}^2 + (\mathit{function\ of}\ X, r).$$

Thus, CBUG is optimally competitive.

Competitiveness of CBUG

(Show Program)
Geometric Progression.

Theorem (Gabriely and Rimon)

NAV_2 has a quadratic universal lower bound, namely given by

$$g_r(x) := \frac{2\pi}{3(1+\pi)^2 r} x^2 \sim \frac{.122x^2}{r}.$$

Theorem (Gabriely and Rimon)

If T is reachable, CBUG solves NAV_2 in time at most

$$\frac{3\pi}{r} l_{opt}^2 + (\textit{function of } X, r).$$

Thus, CBUG is optimally competitive.

Competitiveness of CBUG

(Show Program)
Geometric Progression.

Theorem (Gabriely and Rimon)

NAV_2 has a quadratic universal lower bound, namely given by

$$g_r(x) := \frac{2\pi}{3(1+\pi)^2 r} x^2 \sim \frac{.122x^2}{r}.$$

Theorem (Gabriely and Rimon)

If T is reachable, CBUG solves NAV_2 in time at most

$$\frac{3\pi}{r} l_{opt}^2 + (\textit{function of } X, r).$$

Thus, CBUG is optimally competitive.

Competitiveness of CBUG

(Show Program)
Geometric Progression.

Theorem (Gabriely and Rimon)

NAV_2 has a quadratic universal lower bound, namely given by

$$g_r(x) := \frac{2\pi}{3(1+\pi)^2 r} x^2 \sim \frac{.122x^2}{r}.$$

Theorem (Gabriely and Rimon)

If T is reachable, CBUG solves NAV_2 in time at most

$$\frac{3\pi}{r} l_{opt}^2 + (\textit{function of } X, r).$$

Thus, CBUG is optimally competitive.

Competitiveness of CBUG

(Show Program)
Geometric Progression.

Theorem (Gabriely and Rimon)

NAV_2 has a quadratic universal lower bound, namely given by

$$g_r(x) := \frac{2\pi}{3(1+\pi)^2 r} x^2 \sim \frac{.122x^2}{r}.$$

Theorem (Gabriely and Rimon)

If T is reachable, CBUG solves NAV_2 in time at most

$$\frac{3\pi}{r} l_{opt}^2 + (\textit{function of } X, r).$$

Thus, CBUG is optimally competitive.

Bad News 1

- Proof of second theorem has a flaw....
  - Let $A =$ *area covered*. Gabriely and Rimon assume length of path traversed is at most $\frac{A}{2r}$.
  - Analogue not true for higher dimensions (ex: 3-D r-neighborhood of planar space-filling curve)
- Partial fix comes from Caraballo's Theorem.

Theorem

For every r and every X with finitely many obstacle points there exists k such that the length of the path traversed is at most $k\frac{A}{2r}$.

- Similar results in higher dimensions
- Open question: reasonable result for real environments?

Bad News 1

- Proof of second theorem has a flaw....
  - Let $A =$ *area covered*. Gabriely and Rimon assume length of path traversed is at most $\frac{A}{2r}$.
  - Analogue not true for higher dimensions (ex: 3-D r-neighborhood of planar space-filling curve)
- Partial fix comes from Caraballo's Theorem.

Theorem

For every r and every X with finitely many obstacle points there exists k such that the length of the path traversed is at most $k\frac{A}{2r}$.

- Similar results in higher dimensions
- Open question: reasonable result for real environments?

Bad News 1

- Proof of second theorem has a flaw....
  - Let $A =$ *area covered*. Gabriely and Rimon assume length of path traversed is at most $\frac{A}{2r}$.
  - Analogue not true for higher dimensions (ex: 3-D r-neighborhood of planar space-filling curve)
- Partial fix comes from Caraballo's Theorem.

Theorem

For every r and every X with finitely many obstacle points there exists k such that the length of the path traversed is at most $k\frac{A}{2r}$.

- Similar results in higher dimensions
- Open question: reasonable result for real environments?

Bad News 1

- Proof of second theorem has a flaw....
  - Let $A =$ *area covered*. Gabriely and Rimon assume length of path traversed is at most $\frac{A}{2r}$.
  - Analogue not true for higher dimensions (ex: 3-D r-neighborhood of planar space-filling curve)
- Partial fix comes from Caraballo's Theorem.

Theorem

For every r and every X with finitely many obstacle points there exists k such that the length of the path traversed is at most $k\frac{A}{2r}$.

- Similar results in higher dimensions
- Open question: reasonable result for real environments?

Bad News 1

- Proof of second theorem has a flaw....
  - Let $A =$ *area covered*. Gabriely and Rimon assume length of path traversed is at most $\frac{A}{2r}$.
  - Analogue not true for higher dimensions (ex: 3-D r-neighborhood of planar space-filling curve)
- Partial fix comes from Caraballo's Theorem.

Theorem

For every r and every X with finitely many obstacle points there exists k such that the length of the path traversed is at most $k\frac{A}{2r}$.

- Similar results in higher dimensions
- Open question: reasonable result for real environments?

Bad News 1

- Proof of second theorem has a flaw....
  - Let $A =$ *area covered*. Gabriely and Rimon assume length of path traversed is at most $\frac{A}{2r}$.
  - Analogue not true for higher dimensions (ex: 3-D r-neighborhood of planar space-filling curve)
- Partial fix comes from Caraballo's Theorem.

Theorem

For every r and every X with finitely many obstacle points there exists k such that the length of the path traversed is at most $k\frac{A}{2r}$.

- Similar results in higher dimensions
- Open question: reasonable result for real environments?

Bad News 1

- Proof of second theorem has a flaw....
  - Let $A =$ *area covered*. Gabriely and Rimon assume length of path traversed is at most $\frac{A}{2r}$.
  - Analogue not true for higher dimensions (ex: 3-D r-neighborhood of planar space-filling curve)
- Partial fix comes from Caraballo's Theorem.

Theorem

For every r and every X with finitely many obstacle points there exists k such that the length of the path traversed is at most $k\frac{A}{2r}$.

- Similar results in higher dimensions
- Open question: reasonable result for real environments?

Bad News 2

- $COVER_n$ doesn't make sense for $n > 2$. Neither does an 'efficient' algorithm for NAV_n or $SEARCH_n$:

Theorem (BKS)

If $n \geq 3$, then every algorithm that solves either NAV_n or $SEARCH_n$ is not $O(f)$-competitive for any $f : \mathbb{R} \to \mathbb{R}$.

- Proof via 'parallel corridors' examples:
  - idea: pack as many corridors into as small a volume as possible, so Bob has to potentially explore every one of them.
  - Can pack arbitrarily many corridors into finite volume for $n \geq 3$

Bad News 2

- $COVER_n$ doesn't make sense for $n > 2$. Neither does an 'efficient' algorithm for NAV_n or $SEARCH_n$:

Theorem (BKS)

If $n \geq 3$, then every algorithm that solves either NAV_n or $SEARCH_n$ is not $O(f)$-competitive for any $f : \mathbb{R} \to \mathbb{R}$.

- Proof via 'parallel corridors' examples:
  - idea: pack as many corridors into as small a volume as possible, so Bob has to potentially explore every one of them.
  - Can pack arbitrarily many corridors into finite volume for $n \geq 3$

Bad News 2

- $COVER_n$ doesn't make sense for $n > 2$. Neither does an 'efficient' algorithm for NAV_n or $SEARCH_n$:

Theorem (BKS)

If $n \geq 3$, then every algorithm that solves either NAV_n or $SEARCH_n$ is not $O(f)$-competitive for any $f : \mathbb{R} \to \mathbb{R}$.

- Proof via 'parallel corridors' examples:
  - idea: pack as many corridors into as small a volume as possible, so Bob has to potentially explore every one of them.
  - Can pack arbitrarily many corridors into finite volume for $n \geq 3$

Bad News 2

- $COVER_n$ doesn't make sense for $n > 2$. Neither does an 'efficient' algorithm for NAV_n or $SEARCH_n$:

Theorem (BKS)

If $n \geq 3$, then every algorithm that solves either NAV_n or $SEARCH_n$ is not $O(f)$-competitive for any $f : \mathbb{R} \to \mathbb{R}$.

- Proof via 'parallel corridors' examples:
  - idea: pack as many corridors into as small a volume as possible, so Bob has to potentially explore every one of them.
  - Can pack arbitrarily many corridors into finite volume for $n \geq 3$

Bad News 2

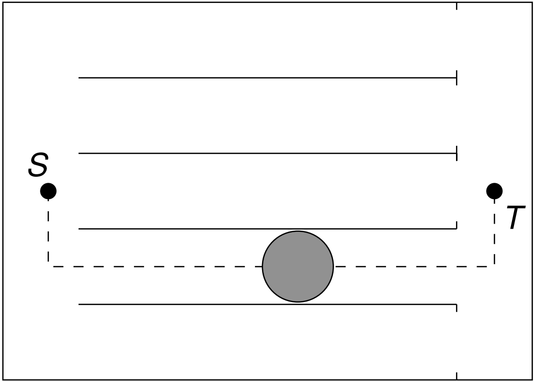
- $COVER_n$ doesn't make sense for $n > 2$. Neither does an 'efficient' algorithm for NAV_n or $SEARCH_n$:

Theorem (BKS)

If $n \geq 3$, then every algorithm that solves either NAV_n or $SEARCH_n$ is not $O(f)$-competitive for any $f : \mathbb{R} \to \mathbb{R}$.

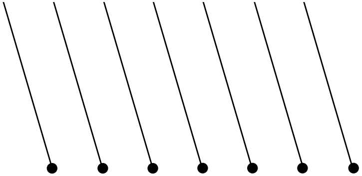
- Proof via 'parallel corridors' examples:
    - idea: pack as many corridors into as small a volume as possible, so Bob has to potentially explore every one of them.
    - Can pack arbitrarily many corridors into finite volume for $n \geq 3$

Parallel Corridors in 2-D

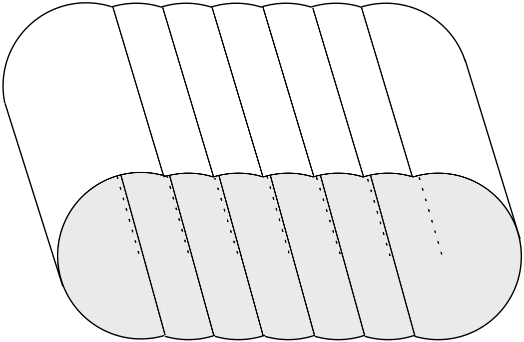


Parallel Corridors in 3-D

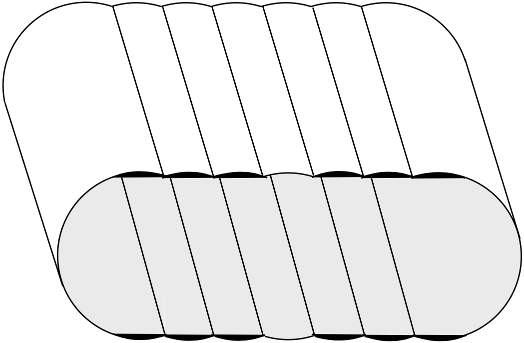
Parallel Corridors in 3-D



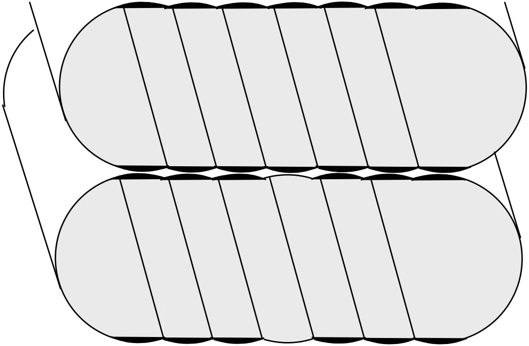
Parallel Corridors in 3-D



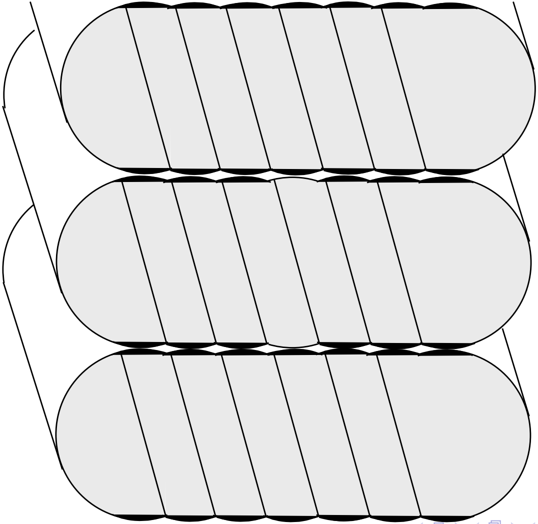
Parallel Corridors in 3-D



Parallel Corridors in 3-D



Parallel Corridors in 3-D



The Fix: Clearance Parameter

- Weaken tasks slightly by adding a *clearance parameter*, $\epsilon > 0$.
- Introduction of ϵ allows us, for instance, to ignore parallel corridor spaces where the corridors are packed too tightly.
- For a fixed ϵ, define

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

and

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

- If a robot of radius r' can occupy two points A and B of X, and $d(A, B) < \kappa$, then Bob can move freely along the straight line from A to B.

The Fix: Clearance Parameter

- Weaken tasks slightly by adding a *clearance parameter*, $\epsilon > 0$.
- Introduction of ϵ allows us, for instance, to ignore parallel corridor spaces where the corridors are packed too tightly.
- For a fixed ϵ, define

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

  and

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

- If a robot of radius r' can occupy two points A and B of X, and $d(A, B) < \kappa$, then Bob can move freely along the straight line from A to B.

The Fix: Clearance Parameter

- Weaken tasks slightly by adding a *clearance parameter*, $\epsilon > 0$.
- Introduction of ϵ allows us, for instance, to ignore parallel corridor spaces where the corridors are packed too tightly.
- For a fixed ϵ, define

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

and

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

- If a robot of radius r' can occupy two points A and B of X, and $d(A, B) < \kappa$, then Bob can move freely along the straight line from A to B.

The Fix: Clearance Parameter

- Weaken tasks slightly by adding a *clearance parameter*, $\epsilon > 0$.
- Introduction of ϵ allows us, for instance, to ignore parallel corridor spaces where the corridors are packed too tightly.
- For a fixed ϵ, define

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

and

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

- If a robot of radius r' can occupy two points A and B of X, and $d(A, B) < \kappa$, then Bob can move freely along the straight line from A to B.

Modifying the Tasks

Definition

The *modified NAV_n and $SEARCH_n$* problems are to reach T if there is an r'-path from S to T, and otherwise reach T or determine no r'-path exists.

Definition

The *modified $COVER_n$* problem is to come within r' of every point within r of an r'-path from S.

- Competitiveness should be measured against the optimal r'-path.

Modifying the Tasks

Definition

The *modified NAV_n and $SEARCH_n$* problems are to reach T if there is an r'-path from S to T, and otherwise reach T or determine no r'-path exists.

Definition

The *modified $COVER_n$* problem is to come within r' of every point within r of an r'-path from S.

- Competitiveness should be measured against the optimal r'-path.

Modifying the Tasks

Definition

The *modified NAV_n and $SEARCH_n$* problems are to reach T if there is an r'-path from S to T, and otherwise reach T or determine no r'-path exists.

Definition

The *modified $COVER_n$* problem is to come within r' of every point within r of an r'-path from S.

- Competitiveness should be measured against the optimal r'-path.

Universal Lower Bound

Linear universal lower bound for $COVER_n$.

Theorem (BKS)

NAV_n and $SEARCH_n$ have a universal lower bound on competitiveness given by

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

- Proof: Analyze parallel corridor spaces (lots of details to check)
- With minor constraints, runtime is at least

$$\frac{l^n_{opt}}{2^{n+2}(1+\sqrt{n-1})^n \kappa^{n-2} r'}.$$

Universal Lower Bound

Linear universal lower bound for $COVER_n$.

Theorem (BKS)

NAV_n and $SEARCH_n$ have a universal lower bound on competitiveness given by

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

- Proof: Analyze parallel corridor spaces (lots of details to check)
- With minor constraints, runtime is at least

$$\frac{l_{opt}^n}{2^{n+2}(1+\sqrt{n-1})^n \kappa^{n-2} r'}.$$

Universal Lower Bound

Linear universal lower bound for $COVER_n$.

Theorem (BKS)

NAV_n and $SEARCH_n$ have a universal lower bound on competitiveness given by

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

- Proof: Analyze parallel corridor spaces (lots of details to check)
- With minor constraints, runtime is at least

$$\frac{l_{opt}^n}{2^{n+2}(1+\sqrt{n-1})^n \kappa^{n-2} r'}.$$

Universal Lower Bound

Linear universal lower bound for $COVER_n$.

Theorem (BKS)

NAV_n and $SEARCH_n$ have a universal lower bound on competitiveness given by

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

- Proof: Analyze parallel corridor spaces (lots of details to check)
- With minor constraints, runtime is at least

$$\frac{l^n_{opt}}{2^{n+2}(1+\sqrt{n-1})^n \kappa^{n-2} r'}.$$

How to Explore Obstacles

- Cannot explore obstacles by touching every point
- Remember explored obstacle points
- Key observations: Only need sufficiently fine mesh of obstacle points: 2 obstacle points of distance less than $2r$ apart prevent Bob from passing between them
- Approximate obstacles by shadow of appropriate Rips complex of sufficiently fine sampling.
- For convenience, we sample via a cubical lattice

How to Explore Obstacles

- Cannot explore obstacles by touching every point
- Remember explored obstacle points
- Key observations: Only need sufficiently fine mesh of obstacle points: 2 obstacle points of distance less than $2r$ apart prevent Bob from passing between them
- Approximate obstacles by shadow of appropriate Rips complex of sufficiently fine sampling.
- For convenience, we sample via a cubical lattice

How to Explore Obstacles

- Cannot explore obstacles by touching every point
- Remember explored obstacle points
- Key observations: Only need sufficiently fine mesh of obstacle points: 2 obstacle points of distance less than $2r$ apart prevent Bob from passing between them
- Approximate obstacles by shadow of appropriate Rips complex of sufficiently fine sampling.
- For convenience, we sample via a cubical lattice

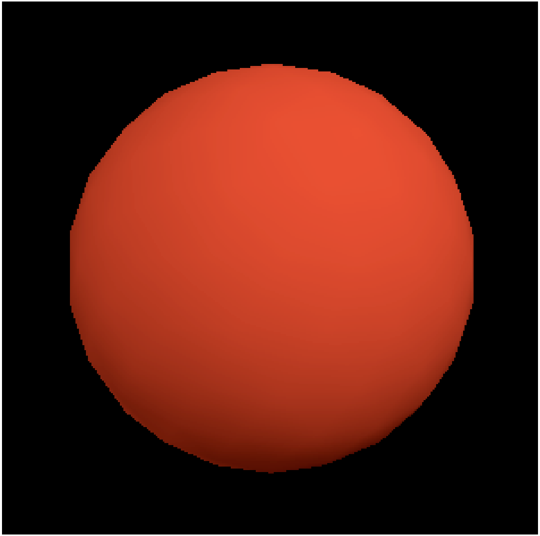
How to Explore Obstacles

- Cannot explore obstacles by touching every point
- Remember explored obstacle points
- Key observations: Only need sufficiently fine mesh of obstacle points: 2 obstacle points of distance less than $2r$ apart prevent Bob from passing between them
- Approximate obstacles by shadow of appropriate Rips complex of sufficiently fine sampling.
- For convenience, we sample via a cubical lattice

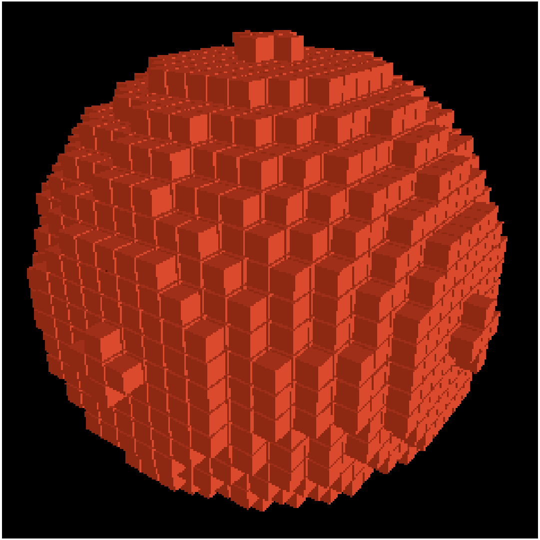
How to Explore Obstacles

- Cannot explore obstacles by touching every point
- Remember explored obstacle points
- Key observations: Only need sufficiently fine mesh of obstacle points: 2 obstacle points of distance less than $2r$ apart prevent Bob from passing between them
- Approximate obstacles by shadow of appropriate Rips complex of sufficiently fine sampling.
- For convenience, we sample via a cubical lattice

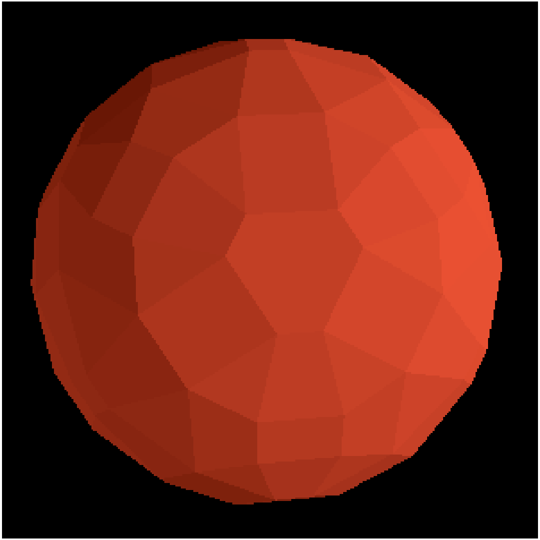
Approximating Obstacles



Approximating Obstacles



Approximating Obstacles



Colors

- White: Unexplored;
- Yellow: Bob's center can be at the center of the cube;
- Red: Too close to an obstacle: the center of a robot of radius $r + \epsilon$ cannot be anywhere in the cube;
- Pink: outside of the virtual boundary.

Colors

- White: Unexplored;
- Yellow: Bob's center can be at the center of the cube;
- Red: Too close to an obstacle: the center of a robot of radius $r + \epsilon$ cannot be anywhere in the cube;
- Pink: outside of the virtual boundary.

Colors

- White: Unexplored;
- Yellow: Bob's center can be at the center of the cube;
- Red: Too close to an obstacle: the center of a robot of radius $r + \epsilon$ cannot be anywhere in the cube;
- Pink: outside of the virtual boundary.

Colors

- White: Unexplored;
- Yellow: Bob's center can be at the center of the cube;
- Red: Too close to an obstacle: the center of a robot of radius $r + \epsilon$ cannot be anywhere in the cube;
- Pink: outside of the virtual boundary.

Solving NAV_n: Boxes

Boxes$_\epsilon$

- Break X into a grid of axis-parallel cubes of side length $l = \min\{\epsilon/2, \epsilon/\sqrt{n}\}$. Color all cubes White.
- Move from S to the center of the current cube, C. Stop if an obstacle is found.
- Define $a_0 = d(S, T') + l$, and set $a = a_0$.
- **While** not in the same cube as T
  - Color cubes outside $\{p : d(S, p) + d(p, T) \leq a\}$ Pink.
  - Explore X using GraphTraverse(C).
  - If no neighbor of a Pink cube is explored, stop.
  - If S is surrounded by points within Red cubes, stop.
  - Double a and color all Pink cubes White.
- Travel towards T. If an obstacle is encountered, stop.

Solving NAV_n: Boxes

Boxes$_\epsilon$

- Break X into a grid of axis-parallel cubes of side length $l = \min\{\epsilon/2, \epsilon/\sqrt{n}\}$. Color all cubes White.
- Move from S to the center of the current cube, C. Stop if an obstacle is found.
- Define $a_0 = d(S, T') + l$, and set $a = a_0$.
- **While** not in the same cube as T
  - Color cubes outside $\{p : d(S, p) + d(p, T) \leq a\}$ Pink.
  - Explore X using GraphTraverse(C).
  - If no neighbor of a Pink cube is explored, stop.
  - If S is surrounded by points within Red cubes, stop.
  - Double a and color all Pink cubes White.
- Travel towards T. If an obstacle is encountered, stop.

Solving NAV_n: Boxes

Boxes$_\epsilon$

- Break X into a grid of axis-parallel cubes of side length $l = \min\{\epsilon/2, \epsilon/\sqrt{n}\}$. Color all cubes White.
- Move from S to the center of the current cube, C. Stop if an obstacle is found.
- Define $a_0 = d(S, T') + l$, and set $a = a_0$.
- **While** not in the same cube as T
  - Color cubes outside $\{p : d(S,p) + d(p,T) \leq a\}$ Pink.
  - Explore X using GraphTraverse(C).
  - If no neighbor of a Pink cube is explored, stop.
  - If S is surrounded by points within Red cubes, stop.
  - Double a and color all Pink cubes White.
- Travel towards T. If an obstacle is encountered, stop.

Solving NAV_n: Boxes

Boxes$_\epsilon$

- Break X into a grid of axis-parallel cubes of side length $l = \min\{\epsilon/2, \epsilon/\sqrt{n}\}$. Color all cubes White.
- Move from S to the center of the current cube, C. Stop if an obstacle is found.
- Define $a_0 = d(S, T') + l$, and set $a = a_0$.
- **While** not in the same cube as T
  - Color cubes outside $\{p : d(S, p) + d(p, T) \leq a\}$ Pink.
  - Explore X using GraphTraverse(C).
  - If no neighbor of a Pink cube is explored, stop.
  - If S is surrounded by points within Red cubes, stop.
  - Double a and color all Pink cubes White.
- Travel towards T. If an obstacle is encountered, stop.

Solving NAV_n: Boxes

Boxes$_\epsilon$

- Break X into a grid of axis-parallel cubes of side length $l = \min\{\epsilon/2, \epsilon/\sqrt{n}\}$. Color all cubes White.
- Move from S to the center of the current cube, C. Stop if an obstacle is found.
- Define $a_0 = d(S, T') + l$, and set $a = a_0$.
- **While** not in the same cube as T
  - Color cubes outside $\{p : d(S,p) + d(p,T) \leq a\}$ Pink.
  - Explore X using GraphTraverse(C).
  - If no neighbor of a Pink cube is explored, stop.
  - If S is surrounded by points within Red cubes, stop.
  - Double a and color all Pink cubes White.
- Travel towards T. If an obstacle is encountered, stop.

Solving NAV_n: Boxes

Boxes$_\epsilon$

- Break X into a grid of axis-parallel cubes of side length $l = \min\{\epsilon/2, \epsilon/\sqrt{n}\}$. Color all cubes White.
- Move from S to the center of the current cube, C. Stop if an obstacle is found.
- Define $a_0 = d(S, T') + l$, and set $a = a_0$.
- **While** not in the same cube as T
  - Color cubes outside $\{p : d(S,p) + d(p,T) \leq a\}$ Pink.
  - Explore X using GraphTraverse(C).
  - If no neighbor of a Pink cube is explored, stop.
  - If S is surrounded by points within Red cubes, stop.
  - Double a and color all Pink cubes White.
- Travel towards T. If an obstacle is encountered, stop.

Solving NAV_n: Boxes

Boxes$_\epsilon$

- Break X into a grid of axis-parallel cubes of side length $l = \min\{\epsilon/2, \epsilon/\sqrt{n}\}$. Color all cubes White.
- Move from S to the center of the current cube, C. Stop if an obstacle is found.
- Define $a_0 = d(S, T') + l$, and set $a = a_0$.
- **While** not in the same cube as T
  - Color cubes outside $\{p : d(S,p) + d(p,T) \leq a\}$ Pink.
  - Explore X using GraphTraverse(C).
  - If no neighbor of a Pink cube is explored, stop.
  - If S is surrounded by points within Red cubes, stop.
  - Double a and color all Pink cubes White.
- Travel towards T. If an obstacle is encountered, stop.

Solving NAV_n: Boxes

Boxes$_\epsilon$

- Break X into a grid of axis-parallel cubes of side length $l = \min\{\epsilon/2, \epsilon/\sqrt{n}\}$. Color all cubes White.
- Move from S to the center of the current cube, C. Stop if an obstacle is found.
- Define $a_0 = d(S, T') + l$, and set $a = a_0$.
- **While** not in the same cube as T
  - Color cubes outside $\{p : d(S, p) + d(p, T) \leq a\}$ Pink.
  - Explore X using GraphTraverse(C).
  - If no neighbor of a Pink cube is explored, stop.
  - If S is surrounded by points within Red cubes, stop.
  - Double a and color all Pink cubes White.
- Travel towards T. If an obstacle is encountered, stop.

GraphTraverse

GraphTraverse(C)

- **If** $T \in C$ Return.
- Set $Adjacent = \{$cubes sharing $(n-1)$-face with $C\}$.
- **While** there are White cubes in $Adjacent$,
  - Pick White $D \in Adjacent$.
  - Move in a straight line toward the center of D.
  - **If** obstacle is hit, color D red and return to center of C.
  - **Else**
    - Color D Yellow
    - GraphTraverse(D,T).
    - **If** T is in the current cube, Return.
  - Travel back to the center of C.
- Return

GraphTraverse

GraphTraverse(C)

- **If** $T \in C$ Return.
- Set $Adjacent = \{$cubes sharing $(n-1)$-face with $C\}$.
- **While** there are White cubes in $Adjacent$,
  - Pick White $D \in Adjacent$.
  - Move in a straight line toward the center of D.
  - **If** obstacle is hit, color D red and return to center of C.
  - **Else**
    - Color D Yellow
    - GraphTraverse(D,T).
    - **If** T is in the current cube, Return.
  - Travel back to the center of C.
- Return

GraphTraverse

GraphTraverse(C)

- **If** $T \in C$ Return.
- Set $Adjacent = \{$cubes sharing $(n-1)$-face with $C\}$.
- **While** there are White cubes in $Adjacent$,
  - Pick White $D \in Adjacent$.
  - Move in a straight line toward the center of D.
  - **If** obstacle is hit, color D red and return to center of C.
  - **Else**
    - Color D Yellow
    - GraphTraverse(D,T).
    - **If** T is in the current cube, Return.
  - Travel back to the center of C.
- Return

GraphTraverse

GraphTraverse(C)

- **If** $T \in C$ Return.
- Set $Adjacent = \{$cubes sharing $(n-1)$-face with $C\}$.
- **While** there are White cubes in $Adjacent$,
  - Pick White $D \in Adjacent$.
  - Move in a straight line toward the center of D.
  - **If** obstacle is hit, color D red and return to center of C.
  - **Else**
    - Color D Yellow
    - GraphTraverse(D,T).
    - **If** T is in the current cube, Return.
  - Travel back to the center of C.
- Return

GraphTraverse

GraphTraverse(C)

- **If** $T \in C$ Return.
- Set $Adjacent = \{$cubes sharing $(n-1)$-face with $C\}$.
- **While** there are White cubes in $Adjacent$,
  - Pick White $D \in Adjacent$.
  - Move in a straight line toward the center of D.
  - **If** obstacle is hit, color D red and return to center of C.
  - **Else**
    - Color D Yellow
    - GraphTraverse(D,T).
    - **If** T is in the current cube, Return.
  - Travel back to the center of C.
- Return

Analysis of Boxes

Theorem (BKS)

If there is an $(r + \epsilon)$-path, p, from S to T, then $Boxes_\epsilon$ will move Bob from S to T.

Sketch of proof:

- points in a cube are at most ϵ apart
- If the center of a robot of radius r' can be SOMEWHERE in a cube, Bob can be ANYWHERE
- Reduces problem to finite graph exploration: stick to 1-skeleton of dual
- GraphTraverse is essentially depth-first search

Analysis of Boxes

Theorem (BKS)

If there is an $(r+\epsilon)$-path, p, from S to T, then $Boxes_\epsilon$ will move Bob from S to T.

Sketch of proof:

- points in a cube are at most ϵ apart
- If the center of a robot of radius r' can be SOMEWHERE in a cube, Bob can be ANYWHERE
- Reduces problem to finite graph exploration: stick to 1-skeleton of dual
- GraphTraverse is essentially depth-first search

Analysis of Boxes

Theorem (BKS)

If there is an $(r+\epsilon)$-path, p, from S to T, then $Boxes_\epsilon$ will move Bob from S to T.

Sketch of proof:

- points in a cube are at most ϵ apart
- If the center of a robot of radius r' can be SOMEWHERE in a cube, Bob can be ANYWHERE
- Reduces problem to finite graph exploration: stick to 1-skeleton of dual
- GraphTraverse is essentially depth-first search

Analysis of Boxes

Theorem (BKS)

If there is an $(r+\epsilon)$-path, p, from S to T, then $Boxes_\epsilon$ will move Bob from S to T.

Sketch of proof:

- points in a cube are at most ϵ apart
- If the center of a robot of radius r' can be SOMEWHERE in a cube, Bob can be ANYWHERE
- Reduces problem to finite graph exploration: stick to 1-skeleton of dual
- GraphTraverse is essentially depth-first search

Analysis of Boxes

Theorem (BKS)

If there is an $(r + \epsilon)$-path, p, from S to T, then $Boxes_\epsilon$ will move Bob from S to T.

Sketch of proof:

- points in a cube are at most ϵ apart
- If the center of a robot of radius r' can be SOMEWHERE in a cube, Bob can be ANYWHERE
- Reduces problem to finite graph exploration: stick to 1-skeleton of dual
- GraphTraverse is essentially depth-first search

Analysis of Boxes

Theorem (BKS)

In spaces without bottlenecks, the length of the path generated by $Boxes_\epsilon$ is at most $c_n(l_{opt})^n\left(\frac{1}{\epsilon}\right)^{n-1} + \frac{d_n}{\epsilon} + \epsilon$ for some constants c_n, d_n depending only on n.

- Sketch of Proof: tree traversal
- Compare to universal lower bound of

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

for small ϵ (recall $\kappa = 2\sqrt{2r\epsilon + \epsilon^2}$).

Analysis of Boxes

Theorem (BKS)

In spaces without bottlenecks, the length of the path generated by $Boxes_\epsilon$ is at most $c_n(l_{opt})^n\left(\frac{1}{\epsilon}\right)^{n-1} + \frac{d_n}{\epsilon} + \epsilon$ for some constants c_n, d_n depending only on n.

- Sketch of Proof: tree traversal
- Compare to universal lower bound of

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

for small ϵ (recall $\kappa = 2\sqrt{2r\epsilon + \epsilon^2}$).

Analysis of Boxes

Theorem (BKS)

In spaces without bottlenecks, the length of the path generated by $Boxes_\epsilon$ is at most $c_n(l_{opt})^n\left(\frac{1}{\epsilon}\right)^{n-1} + \frac{d_n}{\epsilon} + \epsilon$ for some constants c_n, d_n depending only on n.

- Sketch of Proof: tree traversal
- Compare to universal lower bound of

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

for small ϵ (recall $\kappa = 2\sqrt{2r\epsilon + \epsilon^2}$).

Solving *COVER*

Theorem (BKS)

The algorithm CBoxes solves the modified $COVER_n$ problem and is optimally competitive with an upper bound on competitiveness given by $cl_{opt} + d$, where c and d are constants depending on r, n, and ϵ.

Improvements

Our algorithms are bare-bones to prove complexity estimates. Average-case runtimes greatly improve with some modifications:

- Sampling Improvement by using better lattice
- Taking Diagonals Improvement
- Noticing T Improvement to treat *SEARCH* like *NAV*
- Maximal Coloring Improvement
- Disregarding Dead Ends Improvement
- **Greedy Improvement**
- **Subdivision Improvement**

Improvements

Our algorithms are bare-bones to prove complexity estimates. Average-case runtimes greatly improve with some modifications:

- Sampling Improvement by using better lattice
- Taking Diagonals Improvement
- Noticing T Improvement to treat *SEARCH* like *NAV*
- Maximal Coloring Improvement
- Disregarding Dead Ends Improvement
- **Greedy Improvement**
- **Subdivision Improvement**

Improvements

Our algorithms are bare-bones to prove complexity estimates. Average-case runtimes greatly improve with some modifications:

- Sampling Improvement by using better lattice
- Taking Diagonals Improvement
- Noticing T Improvement to treat *SEARCH* like *NAV*
- Maximal Coloring Improvement
- Disregarding Dead Ends Improvement
- **Greedy Improvement**
- **Subdivision Improvement**

Improvements

Our algorithms are bare-bones to prove complexity estimates. Average-case runtimes greatly improve with some modifications:

- Sampling Improvement by using better lattice
- Taking Diagonals Improvement
- Noticing T Improvement to treat *SEARCH* like *NAV*
- Maximal Coloring Improvement
- Disregarding Dead Ends Improvement
- **Greedy Improvement**
- **Subdivision Improvement**

Improvements

Our algorithms are bare-bones to prove complexity estimates. Average-case runtimes greatly improve with some modifications:

- Sampling Improvement by using better lattice
- Taking Diagonals Improvement
- Noticing T Improvement to treat *SEARCH* like *NAV*
- Maximal Coloring Improvement
- Disregarding Dead Ends Improvement
- Greedy Improvement
- Subdivision Improvement

Improvements

Our algorithms are bare-bones to prove complexity estimates. Average-case runtimes greatly improve with some modifications:

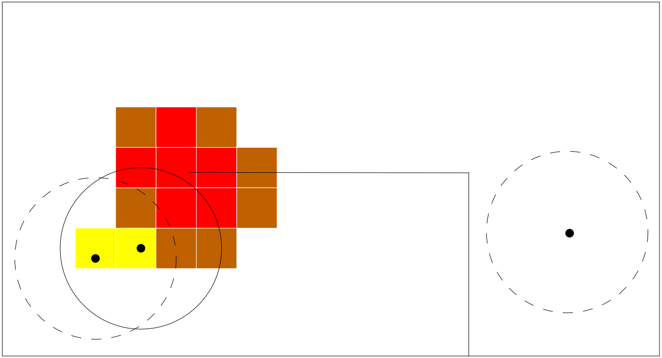
- Sampling Improvement by using better lattice
- Taking Diagonals Improvement
- Noticing T Improvement to treat *SEARCH* like *NAV*
- Maximal Coloring Improvement
- Disregarding Dead Ends Improvement
- **Greedy Improvement**
- Subdivision Improvement

Improvements

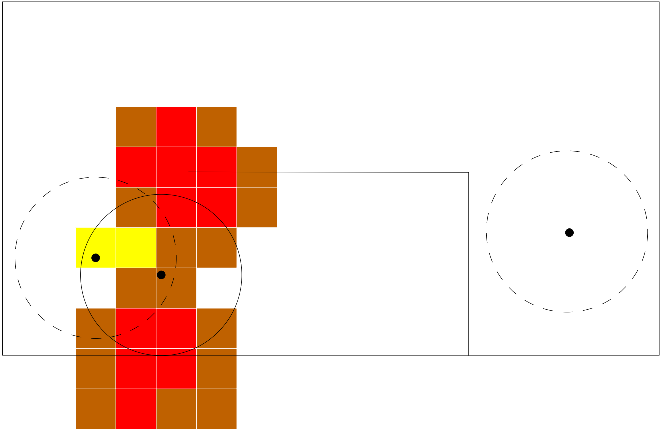
Our algorithms are bare-bones to prove complexity estimates. Average-case runtimes greatly improve with some modifications:

- Sampling Improvement by using better lattice
- Taking Diagonals Improvement
- Noticing T Improvement to treat *SEARCH* like *NAV*
- Maximal Coloring Improvement
- Disregarding Dead Ends Improvement
- **Greedy Improvement**
- **Subdivision Improvement**

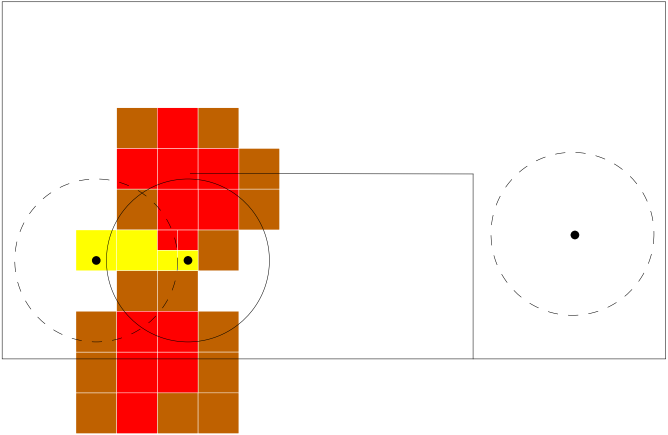
Example of Boxes (improved)



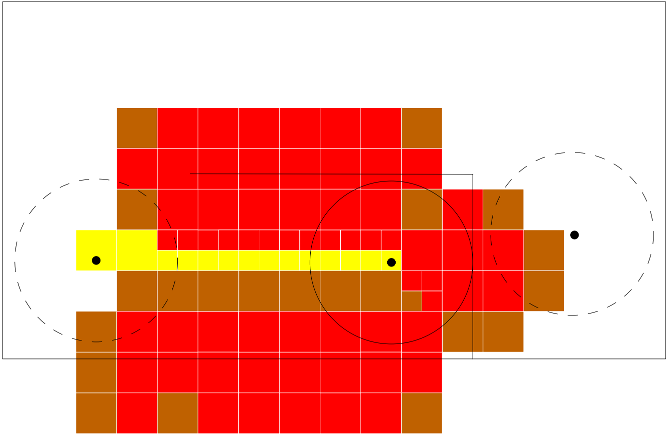
Example of Boxes (improved)



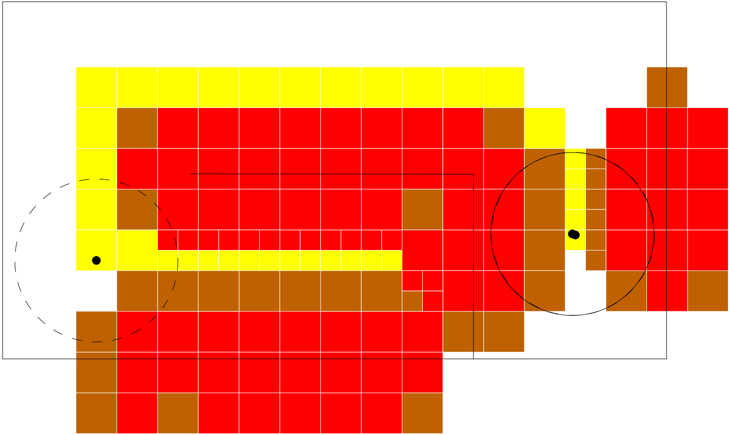
Example of Boxes (improved)



Example of Boxes (improved)



Example of Boxes (improved)



Future Directions

- Improve Boxes to be optimally competitive.
- Calculate average-case complexity with improvements in certain environments
- Analyze case if T is unreachable – 'discompetitive analysis'.
- Program implementation.
- Non-spherical robots, other coordinate systems (tori).
- Consider various applications (arm linkages, Roomba).
- Apply to coordinate-free search and exploration problems (Mars rover).

Future Directions

- Improve Boxes to be optimally competitive.
- Calculate average-case complexity with improvements in certain environments
- Analyze case if T is unreachable – 'discompetitive analysis'.
- Program implementation.
- Non-spherical robots, other coordinate systems (tori).
- Consider various applications (arm linkages, Roomba).
- Apply to coordinate-free search and exploration problems (Mars rover).

Future Directions

- Improve Boxes to be optimally competitive.
- Calculate average-case complexity with improvements in certain environments
- Analyze case if T is unreachable – 'discompetitive analysis'.
- Program implementation.
- Non-spherical robots, other coordinate systems (tori).
- Consider various applications (arm linkages, Roomba).
- Apply to coordinate-free search and exploration problems (Mars rover).

Future Directions

- Improve Boxes to be optimally competitive.
- Calculate average-case complexity with improvements in certain environments
- Analyze case if T is unreachable – 'discompetitive analysis'.
- Program implementation.
- Non-spherical robots, other coordinate systems (tori).
- Consider various applications (arm linkages, Roomba).
- Apply to coordinate-free search and exploration problems (Mars rover).

Future Directions

- Improve Boxes to be optimally competitive.
- Calculate average-case complexity with improvements in certain environments
- Analyze case if T is unreachable – 'discompetitive analysis'.
- Program implementation.
- Non-spherical robots, other coordinate systems (tori).
- Consider various applications (arm linkages, Roomba).
- Apply to coordinate-free search and exploration problems (Mars rover).

Future Directions

- Improve Boxes to be optimally competitive.
- Calculate average-case complexity with improvements in certain environments
- Analyze case if T is unreachable – 'discompetitive analysis'.
- Program implementation.
- Non-spherical robots, other coordinate systems (tori).
- Consider various applications (arm linkages, Roomba).
- Apply to coordinate-free search and exploration problems (Mars rover).

Future Directions

- Improve Boxes to be optimally competitive.
- Calculate average-case complexity with improvements in certain environments
- Analyze case if T is unreachable – 'discompetitive analysis'.
- Program implementation.
- Non-spherical robots, other coordinate systems (tori).
- Consider various applications (arm linkages, Roomba).
- Apply to coordinate-free search and exploration problems (Mars rover).

References I

- Yoav Gabriely and Elon Rimon.
  CBUG: A quadratically competitive mobile robot navigation algorithm.
  Preprint, 2005.

- Vladimir Lumelsky and Alexander Stepanov.
  Path-planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape.
  Algorithmica, 2(4), 403-430, 1987.

- David Caraballo.
  Areas of level sets of distance functions induced by asymmetric norms.
  Pacific J. Math., 218(1), 37-52, 2005.