

# Designing Push Plans for Disk-Shaped Robots

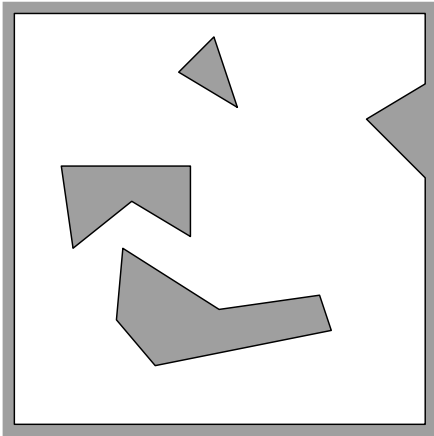
Dirk H.P. Gerrits  
[dirk@dirkgerrits.com](mailto:dirk@dirkgerrits.com)

28 May 2008

- ▶ Context & problem description
- ▶ Our new algorithm
- ▶ New contributions & future research

A fundamental problem in robotics.

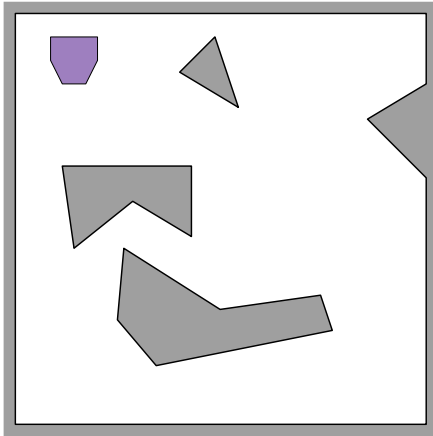
A fundamental problem in robotics.



Given:

- ▶ A collection of **obstacles**.

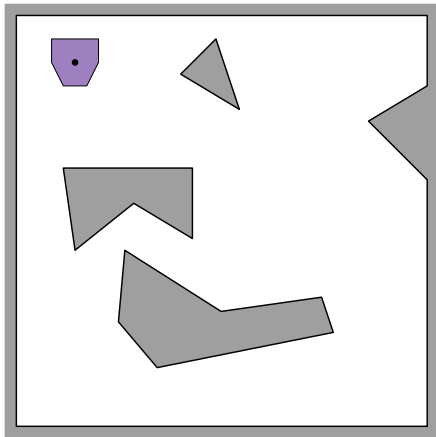
A fundamental problem in robotics.



Given:

- ▶ A collection of **obstacles**.
- ▶ A **robot**

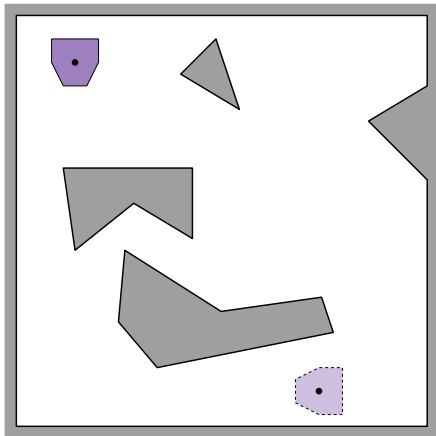
A fundamental problem in robotics.



Given:

- ▶ A collection of **obstacles**.
- ▶ A **robot** at **initial configuration**.

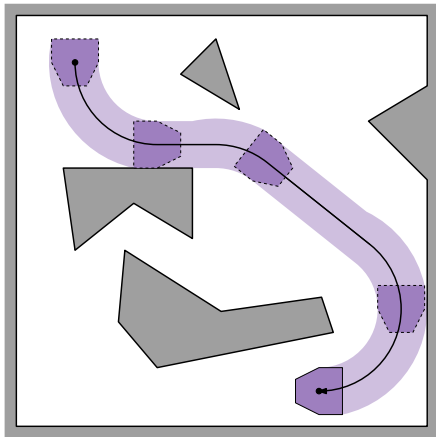
A fundamental problem in robotics.



Given:

- ▶ A collection of **obstacles**.
- ▶ A **robot** at **initial configuration**.
- ▶ A **destination configuration**.

A fundamental problem in robotics.



Given:

- ▶ A collection of **obstacles**.
- ▶ A **robot** at **initial configuration**.
- ▶ A **destination configuration**.

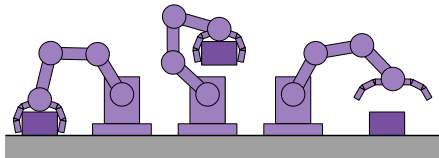
Find:

- ▶ A **collision-free path** from the initial configuration to the destination configuration. (Or report that none exists.)

Now it's not the (active) robot that needs to reach a destination, but a (passive) **object**.

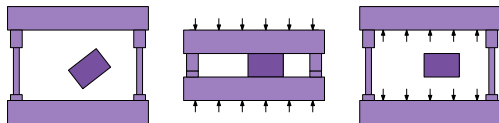
Now it's not the (active) robot that needs to reach a destination, but a (passive) **object**. Many forms of manipulation have been studied:

- ▶ Grasping

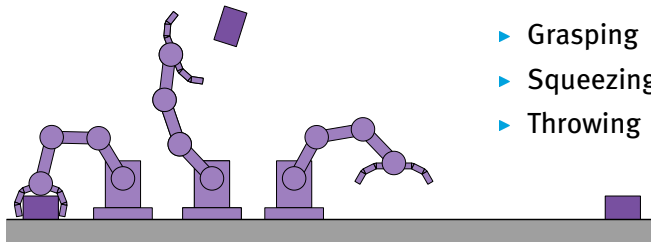


Now it's not the (active) robot that needs to reach a destination, but a (passive) **object**. Many forms of manipulation have been studied:

- ▶ Grasping
- ▶ Squeezing



Now it's not the (active) robot that needs to reach a destination, but a (passive) **object**. Many forms of manipulation have been studied:



- ▶ Grasping
- ▶ Squeezing
- ▶ Throwing

Now it's not the (active) robot that needs to reach a destination, but a (passive) **object**. Many forms of manipulation have been studied:



- ▶ Grasping
- ▶ Squeezing
- ▶ Throwing
- ▶ Pulling

Now it's not the (active) robot that needs to reach a destination, but a (passive) **object**. Many forms of manipulation have been studied:



- ▶ Grasping
- ▶ Squeezing
- ▶ Throwing
- ▶ Pulling
- ▶ Pushing

Now it's not the (active) robot that needs to reach a destination, but a (passive) **object**. Many forms of manipulation have been studied:

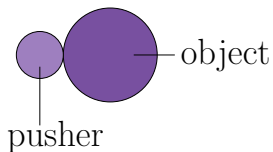


- ▶ Grasping
- ▶ Squeezing
- ▶ Throwing
- ▶ Pulling
- ▶ **Pushing**

Now it's not the (active) robot that needs to reach a destination, but a (passive) **object**. Many forms of manipulation have been studied:

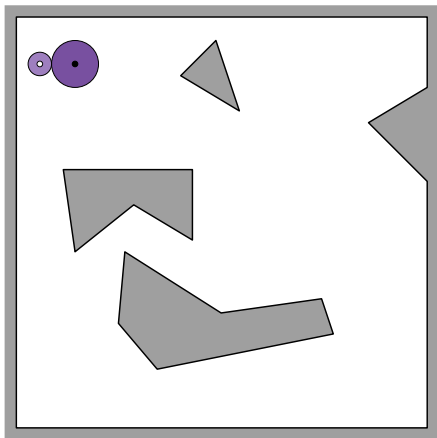


- ▶ Grasping
- ▶ Squeezing
- ▶ Throwing
- ▶ Pulling
- ▶ Pushing



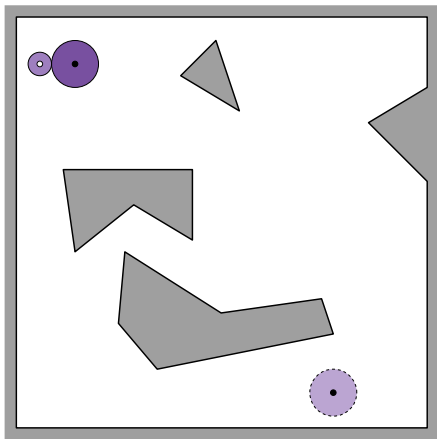
**Given:**

- ▶ Two circular disks in the plane: a smaller **pusher** and a larger **object**.



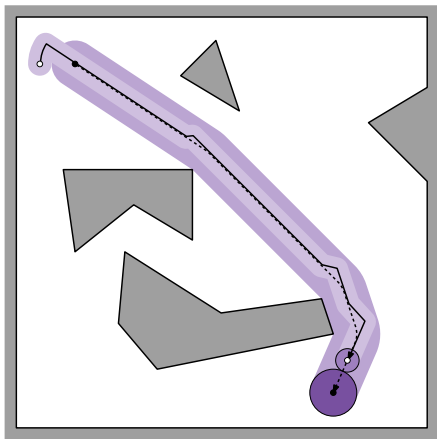
## Given:

- ▶ Two circular disks in the plane: a smaller **pusher** and a larger **object**.
- ▶ A collection of line-segment **obstacles**.



## Given:

- ▶ Two circular disks in the plane: a smaller **pusher** and a larger **object**.
- ▶ A collection of line-segment **obstacles**.
- ▶ A **destination** position for the object.

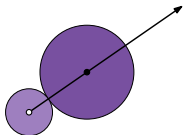


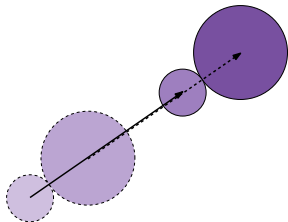
## Given:

- ▶ Two circular disks in the plane: a smaller **pusher** and a larger **object**.
- ▶ A collection of line-segment **obstacles**.
- ▶ A **destination** position for the object.

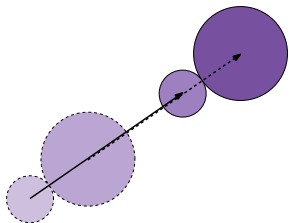
## Find:

- ▶ A path for the pusher to follow (a **push plan**) that makes it push the object to its destination.  
(Or report that none exists.)

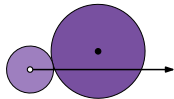


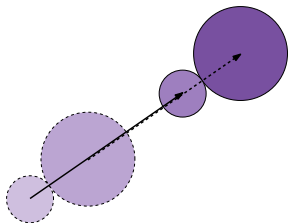


straight line

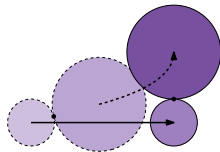


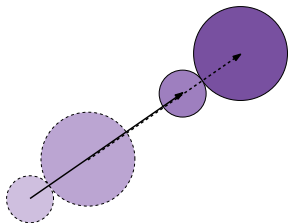
straight line



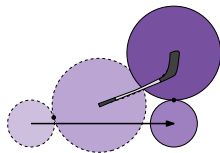


straight line

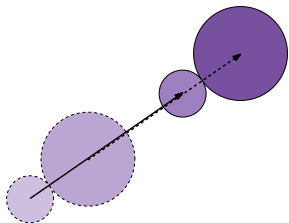




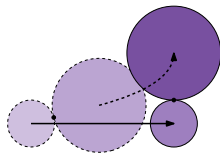
straight line



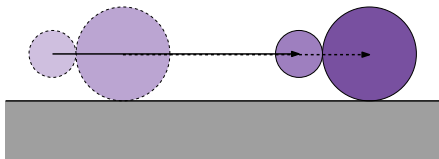
hockey stick

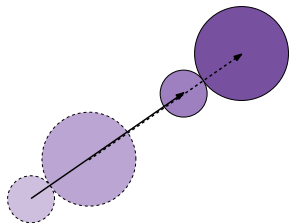


straight line

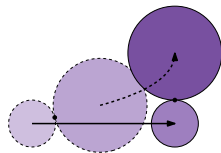


hockey stick

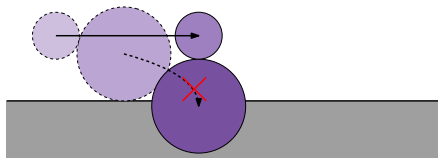


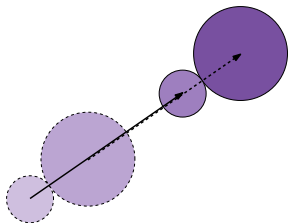


straight line

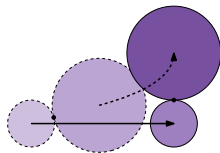


hockey stick



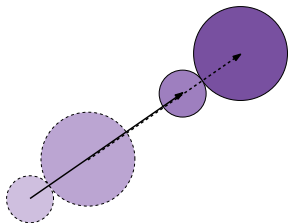


straight line

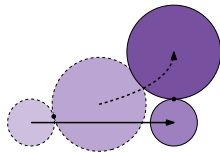


hockey stick

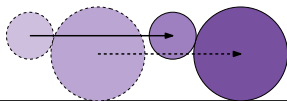




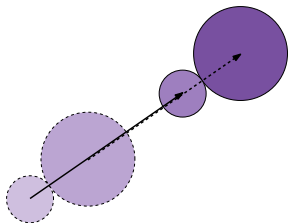
straight-line non-compliant



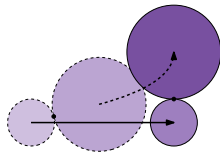
hockey-stick non-compliant



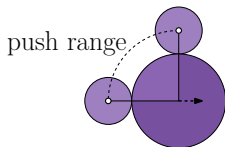
straight-line compliant



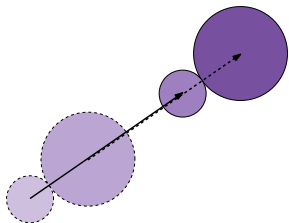
straight-line non-compliant



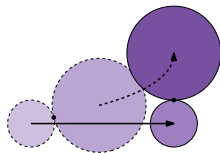
hockey-stick non-compliant



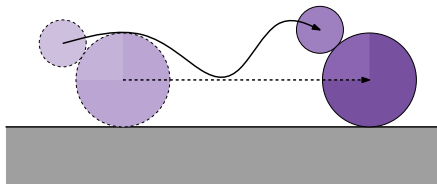
straight-line compliant



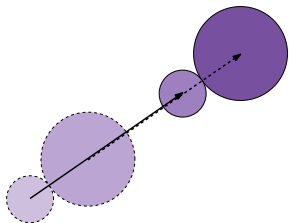
straight-line non-compliant



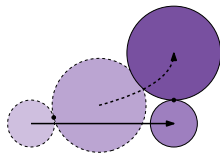
hockey-stick non-compliant



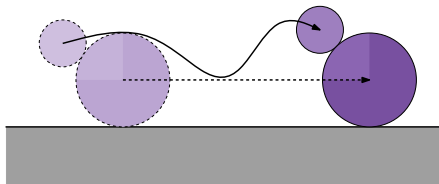
straight-line compliant



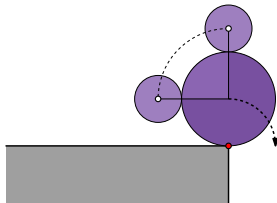
straight-line non-compliant



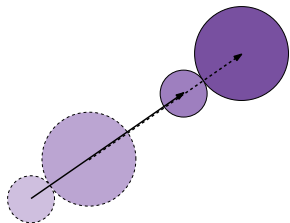
hockey-stick non-compliant



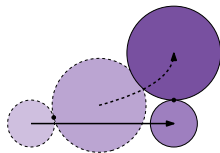
straight-line compliant



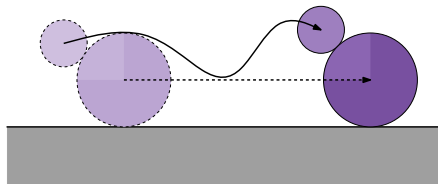
circular compliant



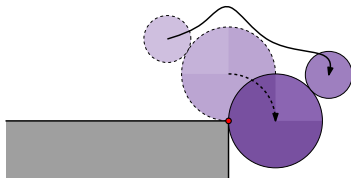
straight-line non-compliant



hockey-stick non-compliant



straight-line compliant



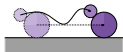
circular compliant



straight-line non-compliant



hockey-stick non-compliant



straight-line compliant



circular compliant

path sections

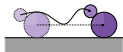
(object moves along curve,  
pusher stays within associated  
push range)



straight-line non-compliant



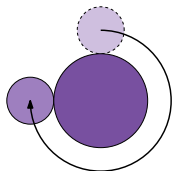
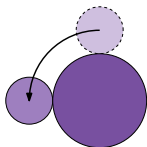
hockey-stick non-compliant



straight-line compliant



circular compliant



path sections

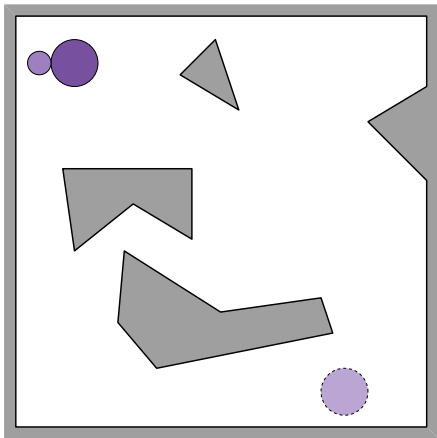
(object moves along curve,  
pusher stays within associated  
push range)

contact transits

(object is stationary,  
pusher turns around it)

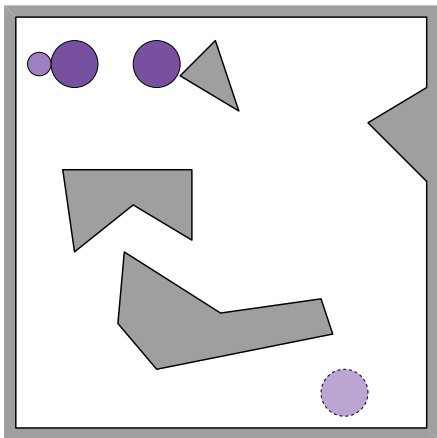
Dennis Nieuwenhuisen designed an algorithm for this problem based on the Rapidly-exploring Random Trees path-planning algorithm.

Dennis Nieuwenhuisen designed an algorithm for this problem based on the Rapidly-exploring Random Trees path-planning algorithm.



Create a tree with the initial configuration as the root:

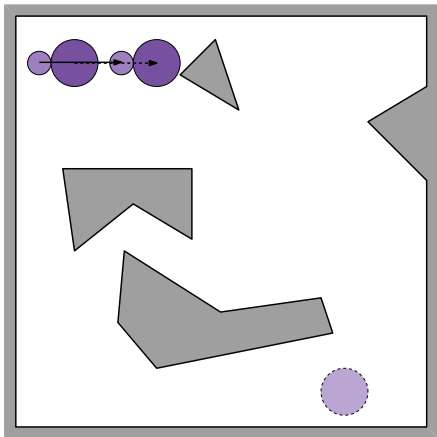
Dennis Nieuwenhuisen designed an algorithm for this problem based on the Rapidly-exploring Random Trees path-planning algorithm.



Create a tree with the initial configuration as the root:

- ▶ Take a random position.

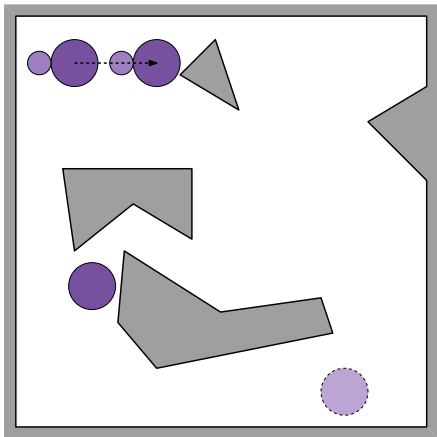
Dennis Nieuwenhuisen designed an algorithm for this problem based on the Rapidly-exploring Random Trees path-planning algorithm.



Create a tree with the initial configuration as the root:

- ▶ Take a random position.
  - ▶ Try a straight line to it from the closest position already in the tree.
- Success → add to tree.

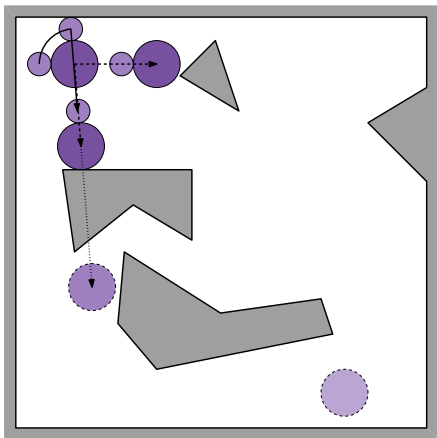
Dennis Nieuwenhuisen designed an algorithm for this problem based on the Rapidly-exploring Random Trees path-planning algorithm.



Create a tree with the initial configuration as the root:

- ▶ Take a random position.
- ▶ Try a straight line to it from the closest position already in the tree.  
**Success** → add to tree.
- ▶ Repeat

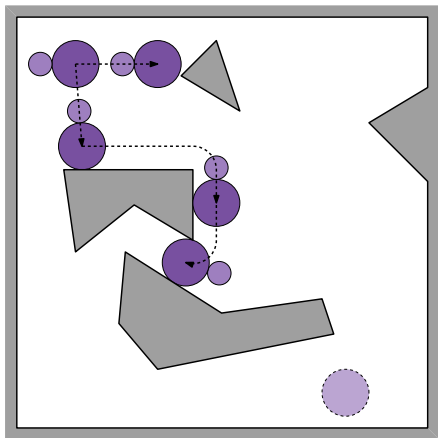
Dennis Nieuwenhuisen designed an algorithm for this problem based on the Rapidly-exploring Random Trees path-planning algorithm.



Create a tree with the initial configuration as the root:

- ▶ Take a random position.
- ▶ Try a straight line to it from the closest position already in the tree.
  - Success → add to tree.
  - Failure
- ▶ Repeat

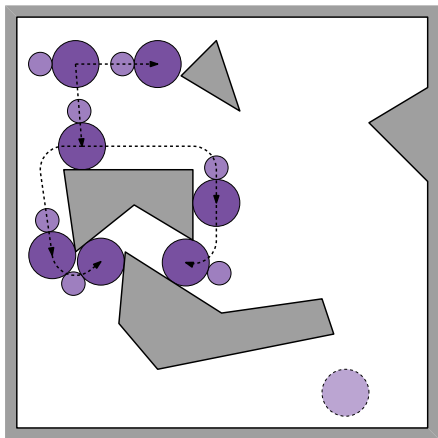
Dennis Nieuwenhuisen designed an algorithm for this problem based on the Rapidly-exploring Random Trees path-planning algorithm.



Create a tree with the initial configuration as the root:

- ▶ Take a random position.
- ▶ Try a straight line to it from the closest position already in the tree.
- ▶ **Success** → add to tree.
- ▶ **Failure** → add reachable compliant positions.
- ▶ Repeat

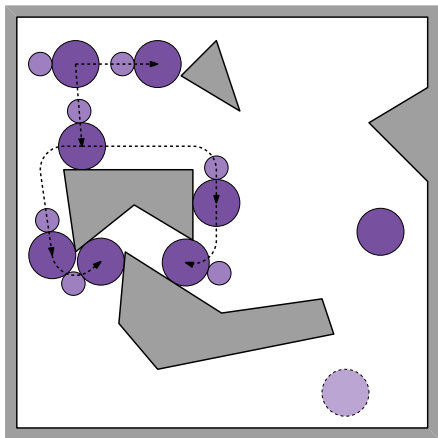
Dennis Nieuwenhuisen designed an algorithm for this problem based on the Rapidly-exploring Random Trees path-planning algorithm.



Create a tree with the initial configuration as the root:

- ▶ Take a random position.
- ▶ Try a straight line to it from the closest position already in the tree.  
**Success** → add to tree.  
**Failure** → add reachable compliant positions.
- ▶ Repeat

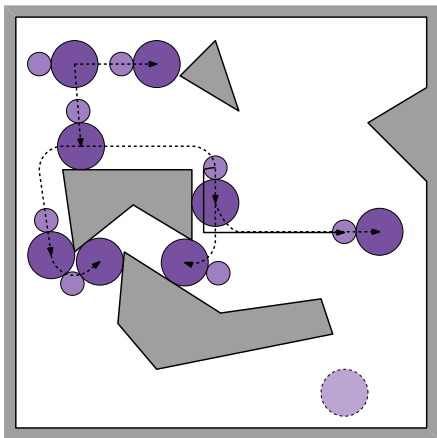
Dennis Nieuwenhuisen designed an algorithm for this problem based on the Rapidly-exploring Random Trees path-planning algorithm.



Create a tree with the initial configuration as the root:

- ▶ Take a random position.
- ▶ Try a straight line to it from the closest position already in the tree.
- ▶ **Success** → add to tree.
- ▶ **Failure** → add reachable compliant positions.
- ▶ Repeat

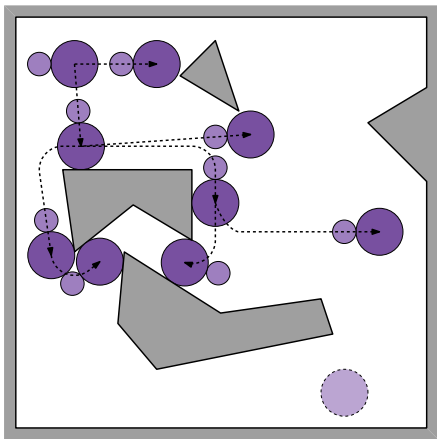
Dennis Nieuwenhuisen designed an algorithm for this problem based on the Rapidly-exploring Random Trees path-planning algorithm.



Create a tree with the initial configuration as the root:

- ▶ Take a random position.
- ▶ Try a straight line to it from the closest position already in the tree.  
**Success** → add to tree.  
**Failure** → add reachable compliant positions.
- ▶ Repeat

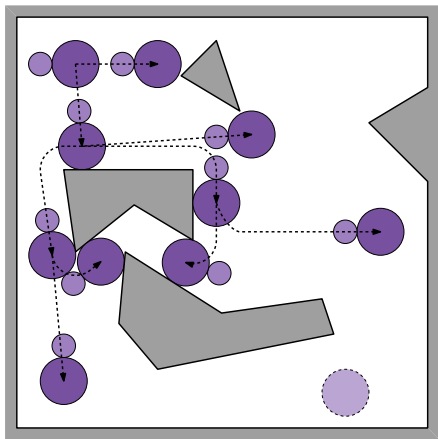
Dennis Nieuwenhuisen designed an algorithm for this problem based on the Rapidly-exploring Random Trees path-planning algorithm.



Create a tree with the initial configuration as the root:

- ▶ Take a random position.
- ▶ Try a straight line to it from the closest position already in the tree.  
**Success** → add to tree.  
**Failure** → add reachable compliant positions.
- ▶ Repeat

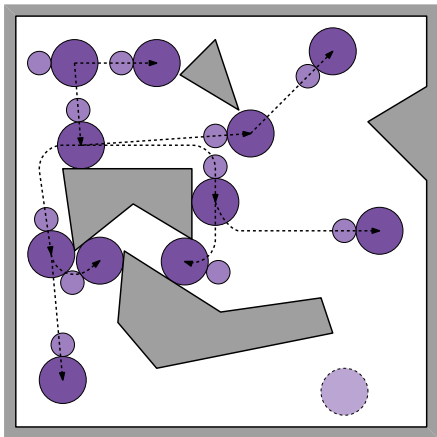
Dennis Nieuwenhuisen designed an algorithm for this problem based on the Rapidly-exploring Random Trees path-planning algorithm.



Create a tree with the initial configuration as the root:

- ▶ Take a random position.
- ▶ Try a straight line to it from the closest position already in the tree.  
**Success** → add to tree.  
**Failure** → add reachable compliant positions.
- ▶ Repeat

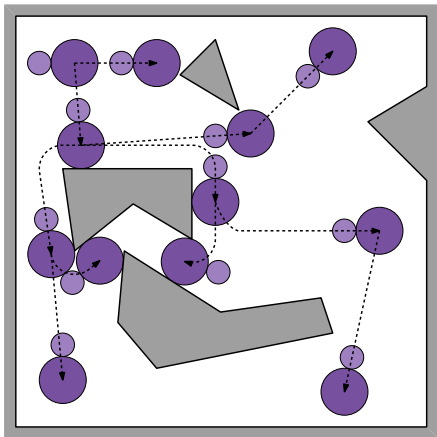
Dennis Nieuwenhuisen designed an algorithm for this problem based on the Rapidly-exploring Random Trees path-planning algorithm.



Create a tree with the initial configuration as the root:

- ▶ Take a random position.
- ▶ Try a straight line to it from the closest position already in the tree.
- ▶ **Success** → add to tree.
- ▶ **Failure** → add reachable compliant positions.
- ▶ Repeat

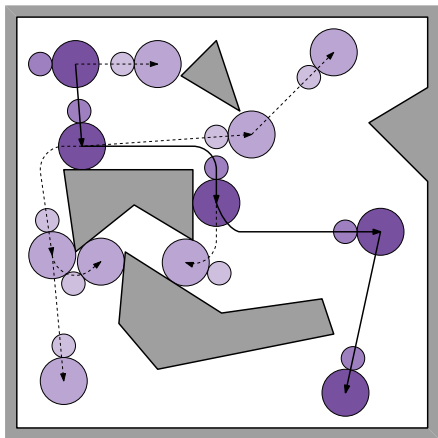
Dennis Nieuwenhuisen designed an algorithm for this problem based on the Rapidly-exploring Random Trees path-planning algorithm.



Create a tree with the initial configuration as the root:

- ▶ Take a random position.
- ▶ Try a straight line to it from the closest position already in the tree.  
**Success** → add to tree.  
**Failure** → add reachable compliant positions.
- ▶ Repeat until destination gets added to the tree (or other stopping criterion).

Dennis Nieuwenhuisen designed an algorithm for this problem based on the Rapidly-exploring Random Trees path-planning algorithm.



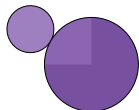
Create a tree with the initial configuration as the root:

- ▶ Take a random position.
- ▶ Try a straight line to it from the closest position already in the tree.  
**Success** → add to tree.  
**Failure** → add reachable compliant positions.
- ▶ Repeat until destination gets added to the tree (or other stopping criterion).

Nieuwenhuisen's algorithm needs a subroutine for this subproblem:

Given:

- ▶ Two circular disks in the plane: a smaller **pusher** and a larger **object**.



Nieuwenhuisen's algorithm needs a subroutine for this subproblem:

Given:

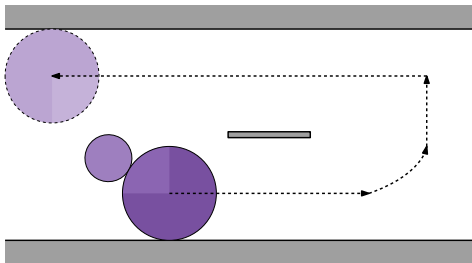


- ▶ Two circular disks in the plane: a smaller **pusher** and a larger **object**.
- ▶ A collection of line-segment **obstacles**.



Nieuwenhuisen's algorithm needs a subroutine for this subproblem:

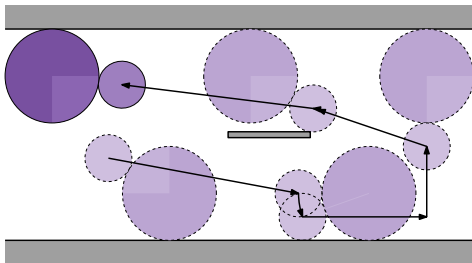
Given:



- ▶ Two circular disks in the plane: a smaller **pusher** and a larger **object**.
- ▶ A collection of line-segment **obstacles**.
- ▶ A path of obstacle-free **path sections** for the object.

Nieuwenhuisen's algorithm needs a subroutine for this subproblem:

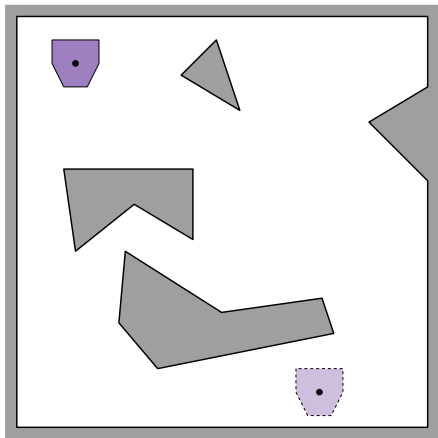
Given:



- ▶ Two circular disks in the plane: a smaller **pusher** and a larger **object**.
- ▶ A collection of line-segment **obstacles**.
- ▶ A path of obstacle-free **path sections** for the object.

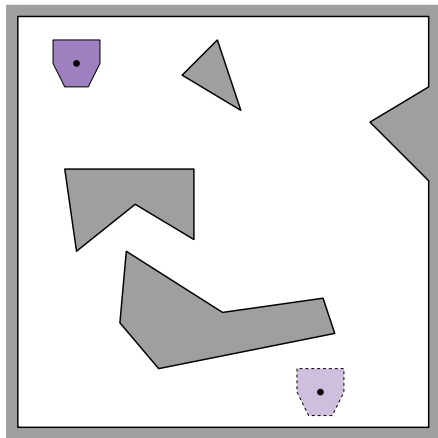
Find:

- ▶ A **push plan** that makes the pusher push the object (as far as possible) along its path.

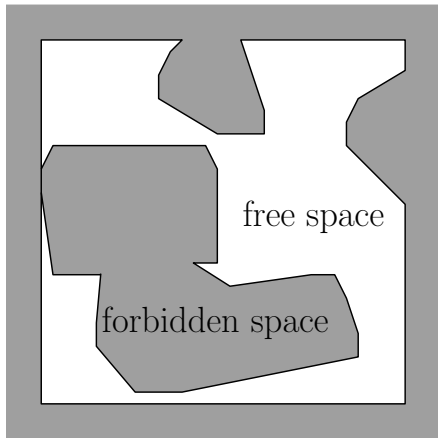




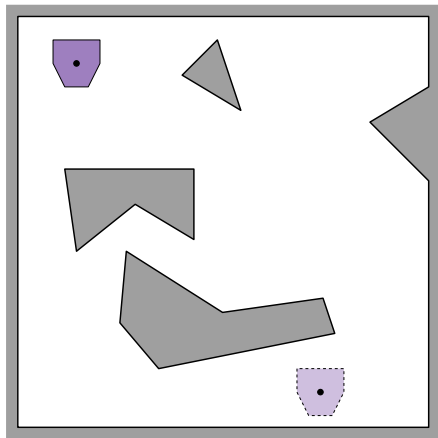




work space

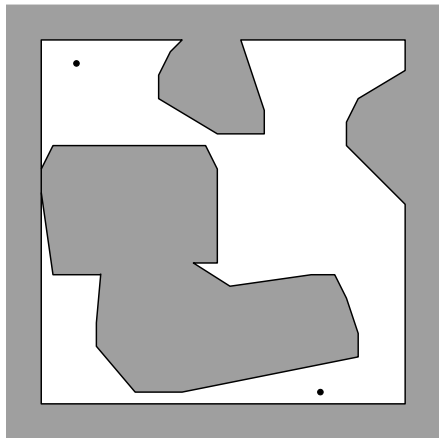


configuration space



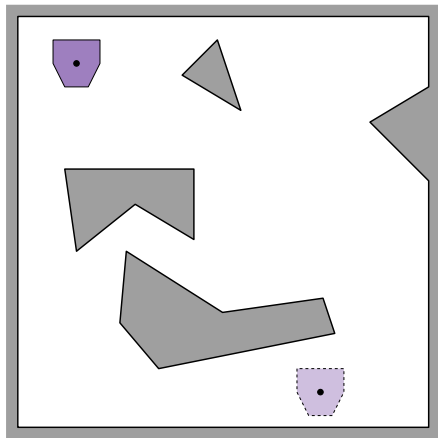
work space

configuration = 2D position



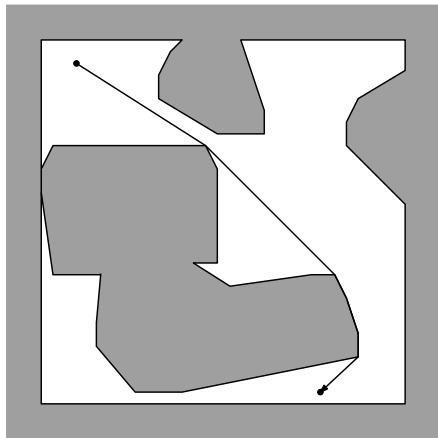
configuration space

point in 2D



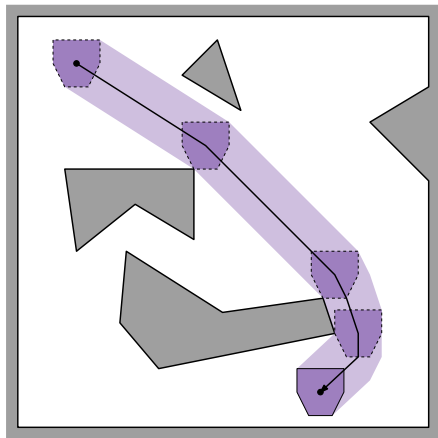
work space

configuration = 2D position



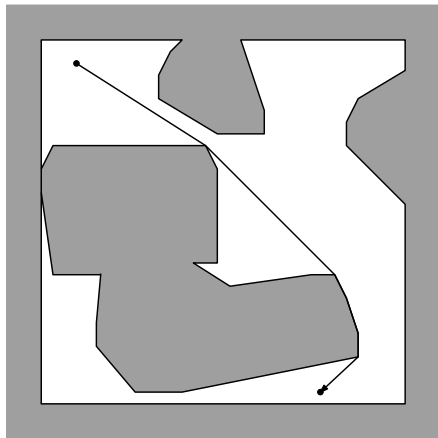
configuration space

point in 2D



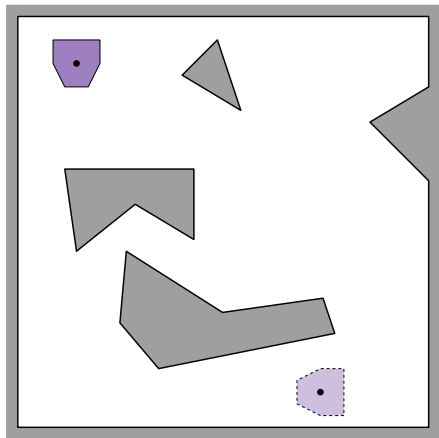
work space

configuration = 2D position



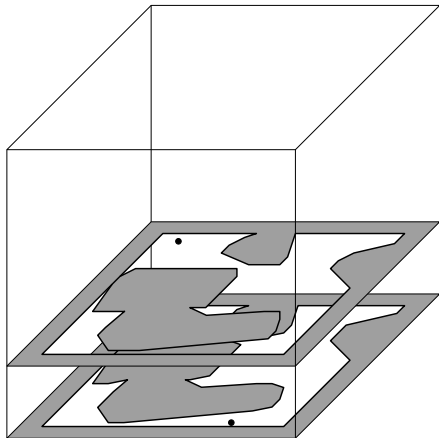
configuration space

point in 2D



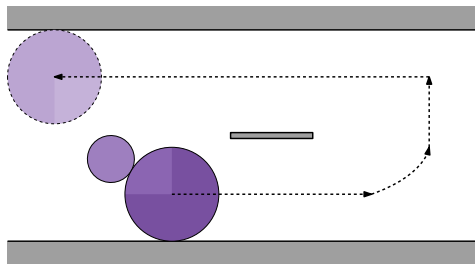
work space

configuration = 2D position +  
1D orientation



configuration space

point in 3D



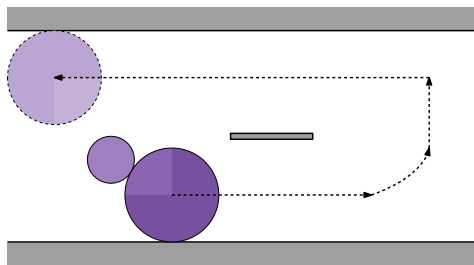
work space

configuration =  
2D object position +  
2D pusher position



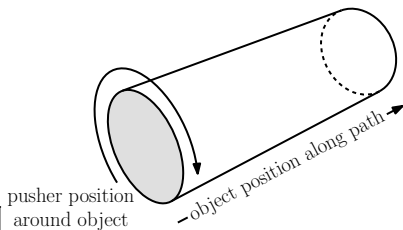
configuration space

point in 4D ?!



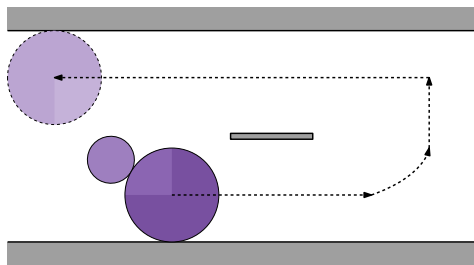
work space

configuration =  
1D object position **along path** +  
1D pusher position **around object**



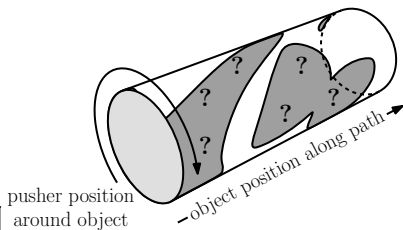
configuration space

point in 2D (on cylinder)



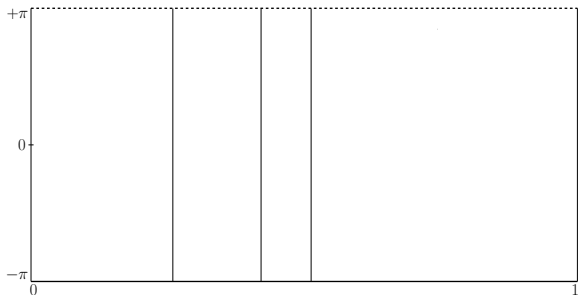
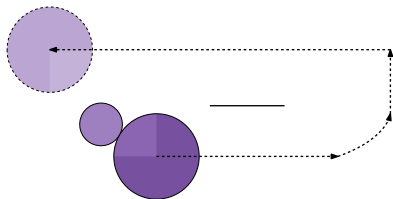
work space

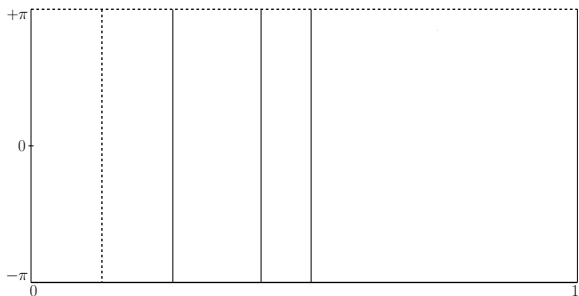
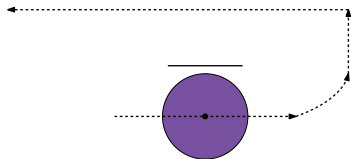
configuration =  
1D object position **along path** +  
1D pusher position **around object**

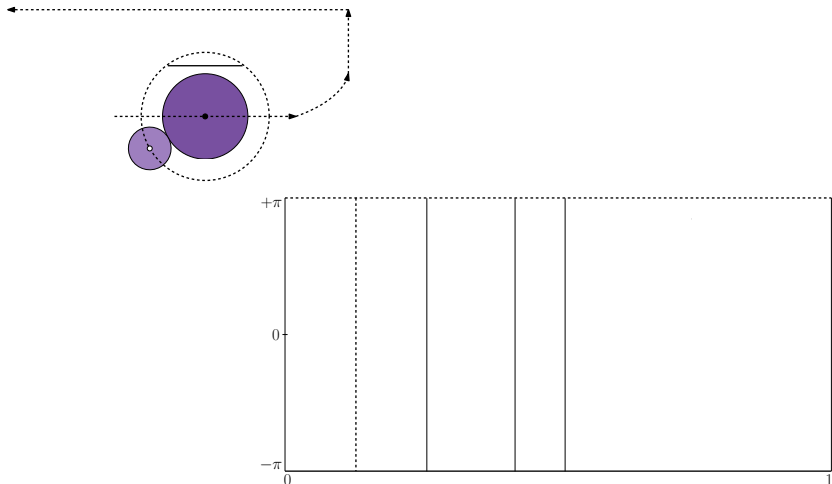


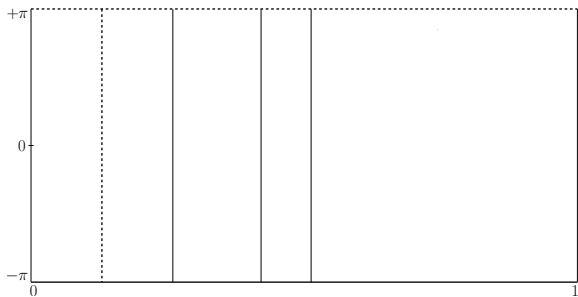
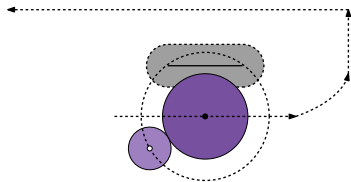
configuration space

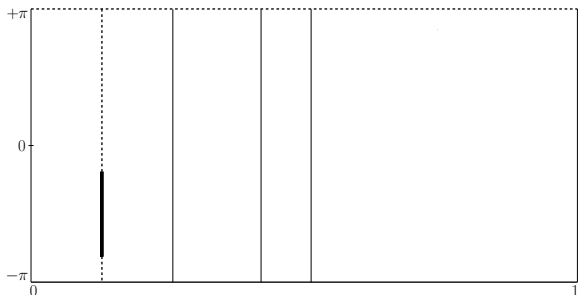
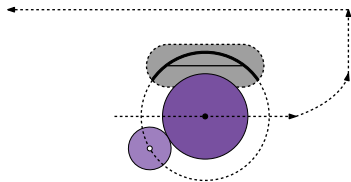
point in 2D (on cylinder)

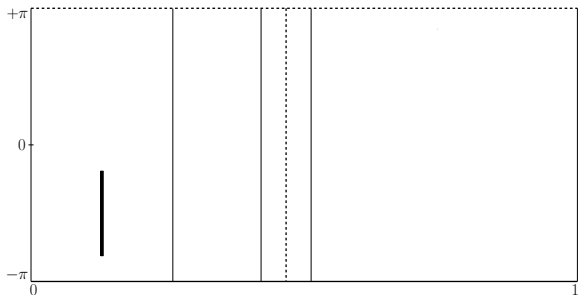
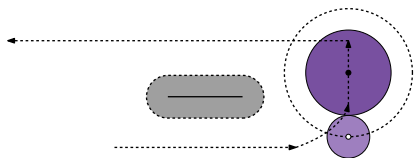


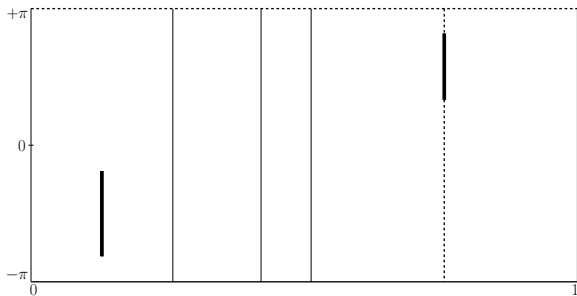
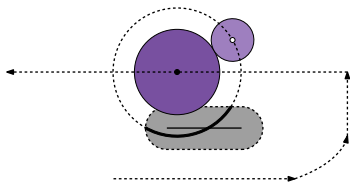


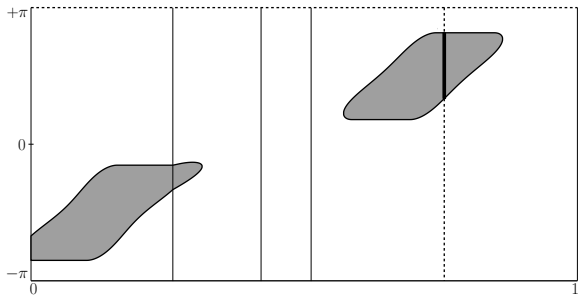
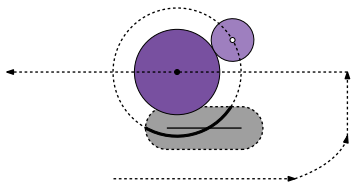


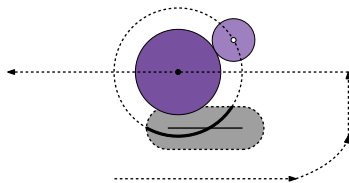






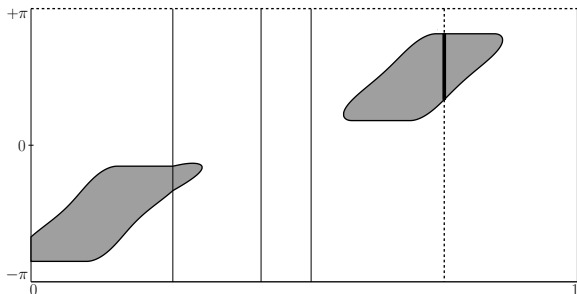


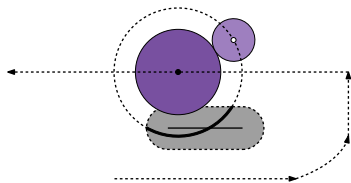




One  $\mathcal{C}$ -space obstacle:

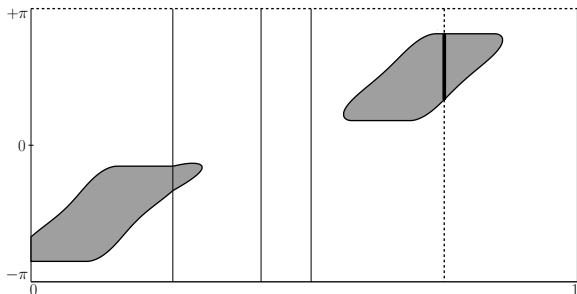
- ▶ can have multiple components

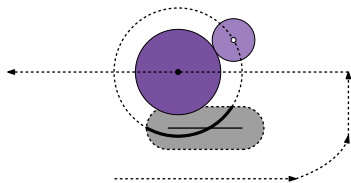




One  $\mathcal{C}$ -space obstacle:

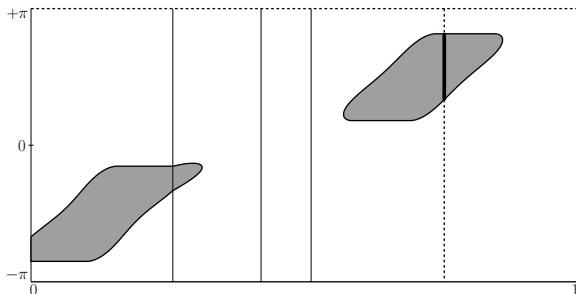
- ▶ can have multiple components, but total complexity  $O(\#sections)$

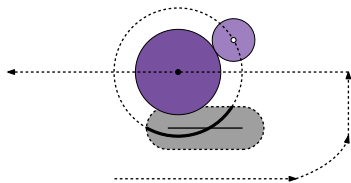




One  $\mathcal{C}$ -space obstacle:

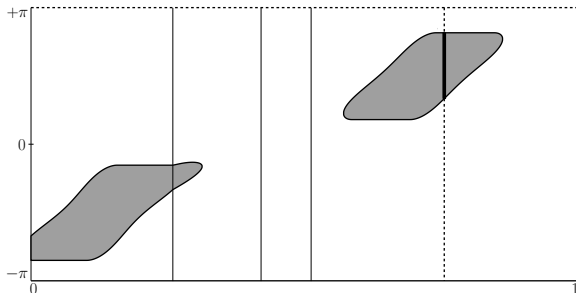
- ▶ can have multiple components, but total complexity  $O(\#sections)$
- ▶ “weird” curves

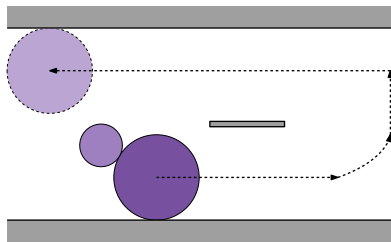




One  $\mathcal{C}$ -space obstacle:

- ▶ can have multiple components, but total complexity  $O(\#sections)$
- ▶ “weird” curves, but can be handled exactly with smart representation

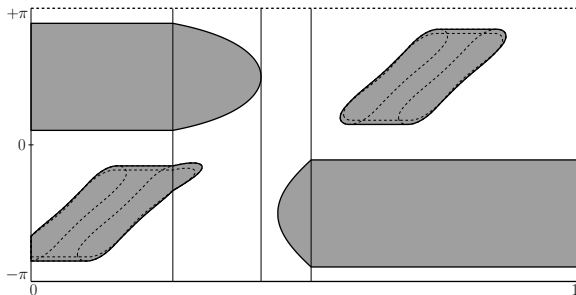


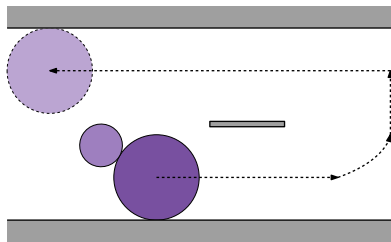


One  $\mathcal{C}$ -space obstacle:

- ▶ can have multiple components, but total complexity  $O(\#\text{sections})$
- ▶ “weird” curves, but can be handled exactly with smart representation

Union of all  $\mathcal{C}$ -space obstacles



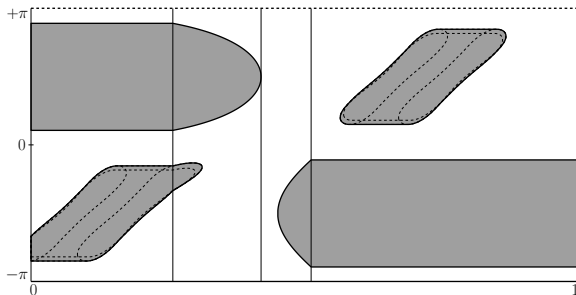


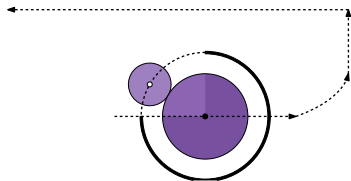
One  $\mathcal{C}$ -space obstacle:

- ▶ can have multiple components, but total complexity  $O(\#sections)$
- ▶ “weird” curves, but can be handled exactly with smart representation

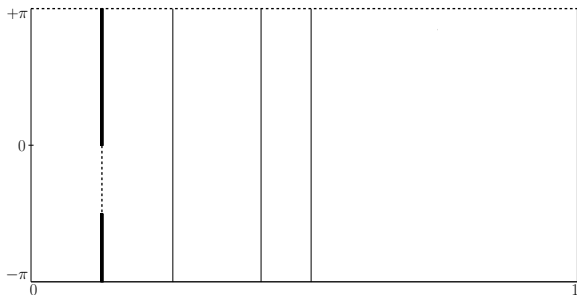
Union of all  $\mathcal{C}$ -space obstacles:

- ▶ linear complexity:  
 $O(\#obstacles \times \#sections)$

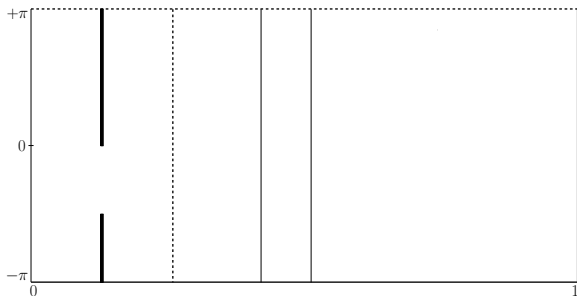
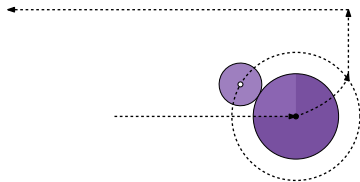




A configuration is also in the forbidden space if the pusher is outside the push range.

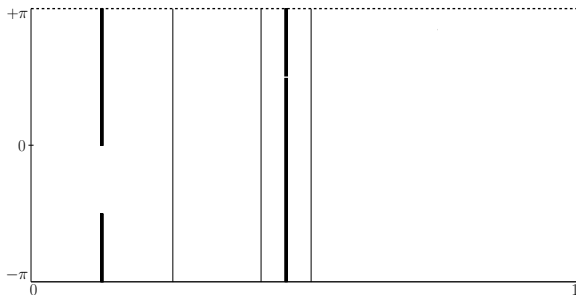


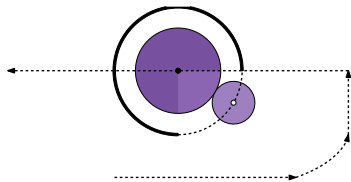
A configuration is also in the forbidden space if the pusher is outside the push range.



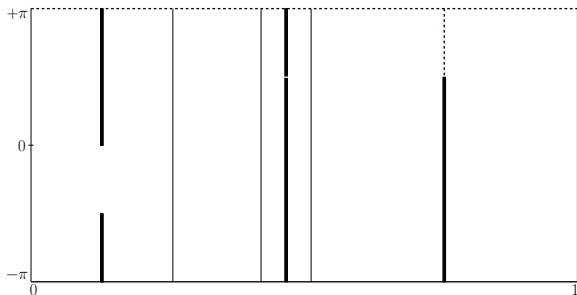


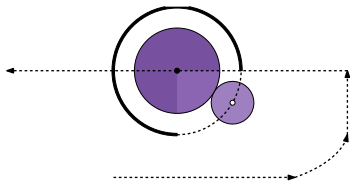
A configuration is also in the forbidden space if the pusher is outside the push range.



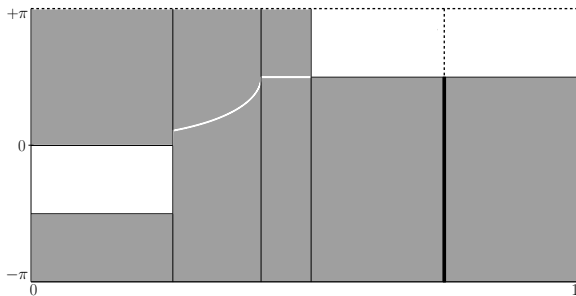


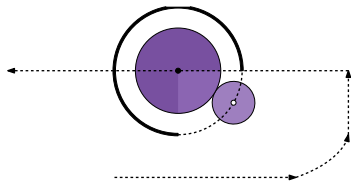
A configuration is also in the forbidden space if the pusher is outside the push range.





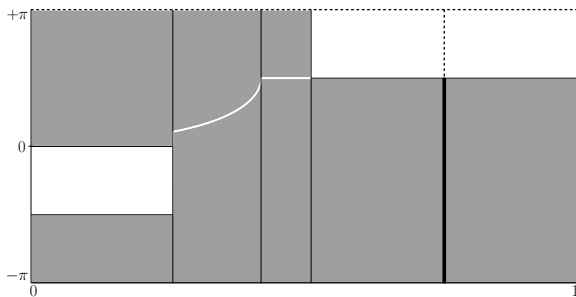
A configuration is also in the forbidden space if the pusher is outside the push range.

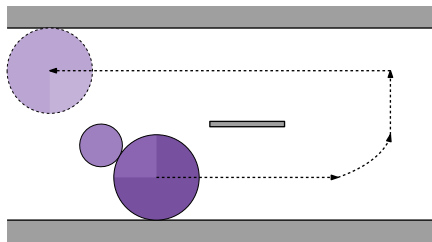




A configuration is also in the forbidden space if the pusher is outside the push range.

Forbidden push range also has  $O(\#sections)$  complexity.

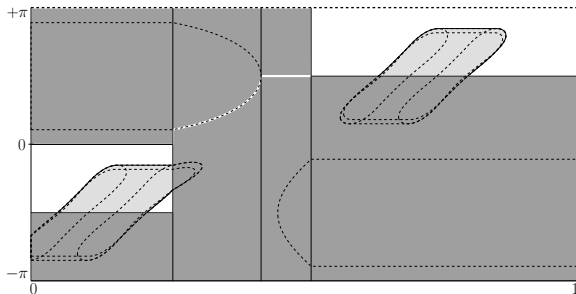


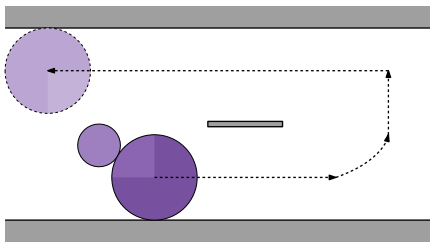


A configuration is also in the forbidden space if the pusher is outside the push range.

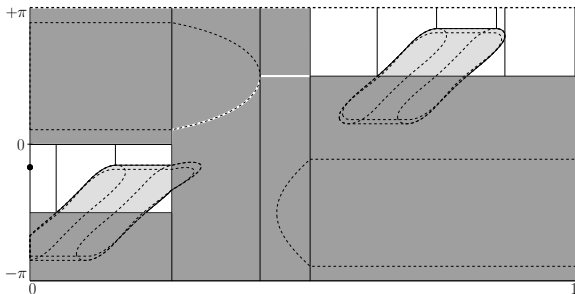
Forbidden push range also has  $O(\#sections)$  complexity.

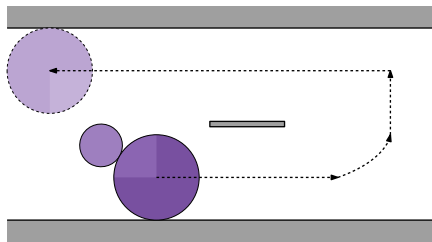
Total forbidden space:  
 $O(\#obstacles \times$   
 $\#sections)$



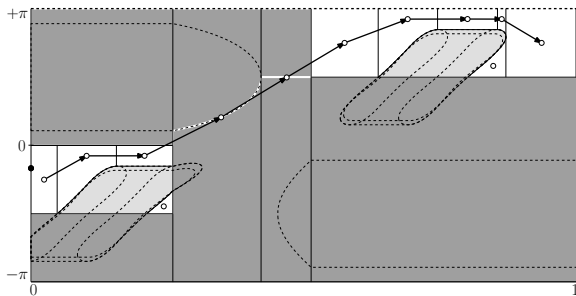


- ▶ Create vertical decomposition of the free space

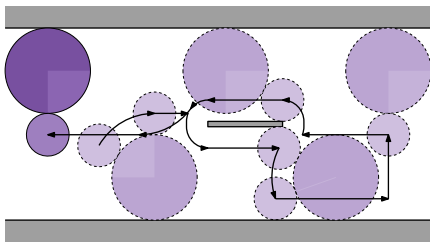




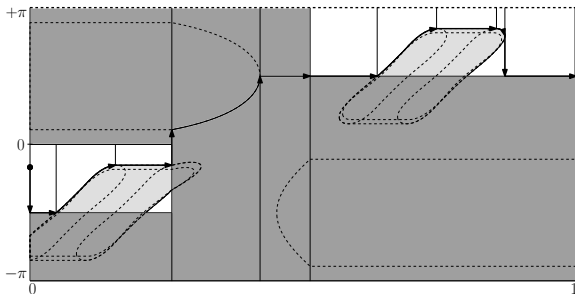
- ▶ Create vertical decomposition of the free space
- ▶ Find path through graph induced by cells

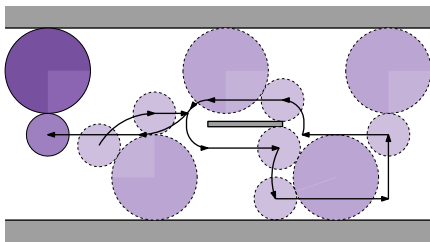






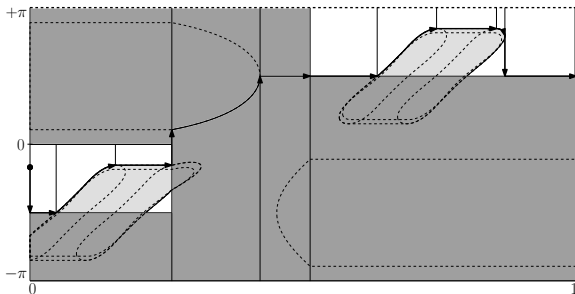
- ▶ Create vertical decomposition of the free space
- ▶ Find path through graph induced by cells
- ▶ Follow boundaries of those cells

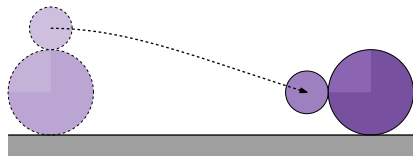




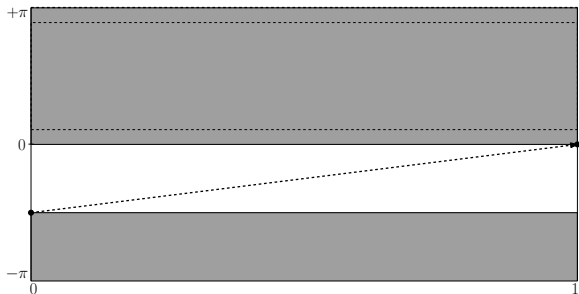
- ▶ Create vertical decomposition of the free space
- ▶ Find path through graph induced by cells
- ▶ Follow boundaries of those cells

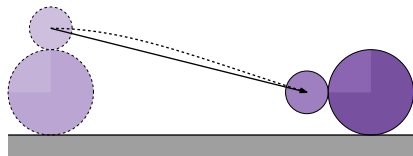
Fast method, but it gives a “bad” push plan. Can we find something shorter?



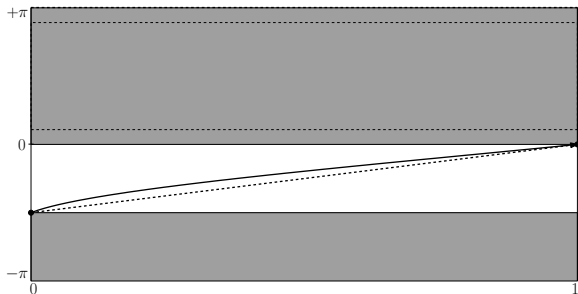


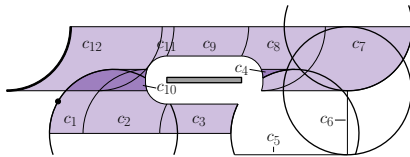
To minimize the distance traveled by the pusher, take the shortest path through free space?



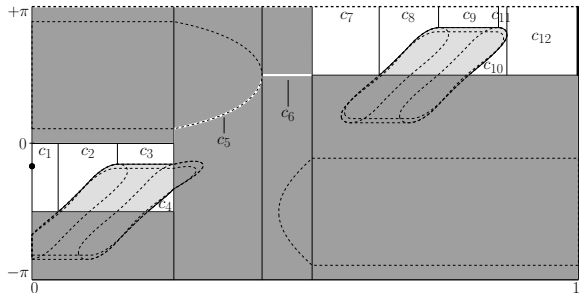


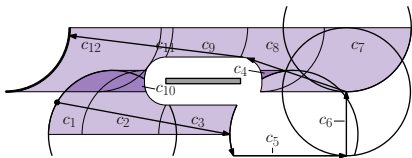
To minimize the distance traveled by the pusher, take the shortest path through free space? **No!**



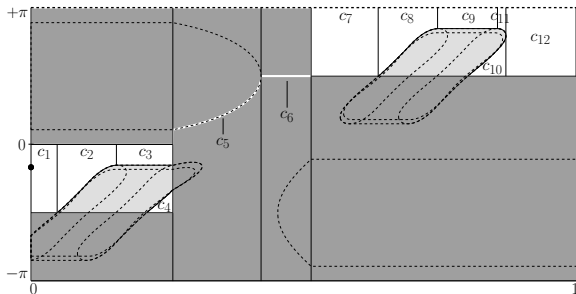


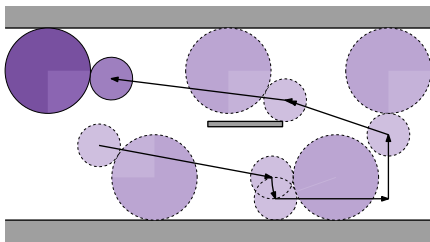
- ▶ Compute the area of allowed pusher positions for each cell



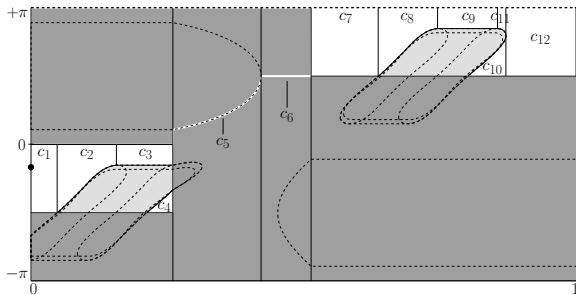


- ▶ Compute the area of allowed pusher positions for each cell
- ▶ Find a shortest path through these work-space cells





- ▶ Compute the area of allowed pusher positions for each cell
- ▶ Find a shortest path through these work-space cells



- ▶ Our algorithm for contact-preserving push plans is more general:
  - it can solve the pulling problem

- ▶ Our algorithm for contact-preserving push plans is more general:
  - it can solve the pulling problem
  - it can handle non-line-segment obstacles

- ▶ Our algorithm for contact-preserving push plans is more general:
  - it can solve the pulling problem
  - it can handle non-line-segment obstacles
  - it can handle a pusher larger than the object

- ▶ Our algorithm for contact-preserving push plans is more general:
  - it can solve the pulling problem
  - it can handle non-line-segment obstacles
  - it can handle a pusher larger than the objectwhile having a better running time!

- ▶ Our algorithm for contact-preserving push plans is more general while having a better running time!

	High obstacle density		Low obstacle density	
	Nieuwenh.	Our method	Nieuwenhuisen	Our method
Preprocess	$n^2 \log n$	$n \log n$ (*)	$n^2 \log n$	$n \log n$ (*)
Any CPPP	$kn \log n$	$kn \log n$ (*)	$(k+n) \log(k+n)$	$(k+n) \log(k+n)$

(\*) These entries are expected times. For the worst-case times, replace  $\log n$  by  $\log^2 n$ .

- ▶ Our algorithm for contact-preserving push plans is more general while having a better running time!
- ▶ Can also compute shortest contact-preserving push plans

	High obstacle density		Low obstacle density	
	Nieuwenh.	Our method	Nieuwenhuisen	Our method
Preprocess	$n^2 \log n$	$n \log n$ (*)	$n^2 \log n$	$n \log n$ (*)
Any CPPP	$kn \log n$	$kn \log n$ (*)	$(k+n) \log(k+n)$	$(k+n) \log(k+n)$
A shortest CPPP	—	$k^2 n^2 \log(kn)$	—	$(k+n) \log(k+n) + k^2 \log k$ (**)

(\*) These entries are expected times. For the worst-case times, replace  $\log n$  by  $\log^2 n$ .

(\*\*) This yields a “quasi-optimal” solution.

- ▶ Our algorithm for contact-preserving push plans is more general while having a better running time!
- ▶ Can also compute shortest contact-preserving push plans
- ▶ Can also compute unrestricted push plans, which we proved is sometimes necessary

	High obstacle density		Low obstacle density	
	Nieuwenh.	Our method	Nieuwenhuisen	Our method
Preprocess	$n^2 \log n$	$n \log n$ (*)	$n^2 \log n$	$n \log n$ (*)
Any CPPP	$kn \log n$	$kn \log n$ (*)	$(k+n) \log(k+n)$	$(k+n) \log(k+n)$
A shortest CPPP	—	$k^2 n^2 \log(kn)$	—	$(k+n) \log(k+n) + k^2 \log k$ (**)
Any UPP	—	$kn \log(kn) + kn^2 \log n$	—	$(k+n) \log(k+n) + kn$

(\*) These entries are expected times. For the worst-case times, replace  $\log n$  by  $\log^2 n$ .

(\*\*) This yields a “quasi-optimal” solution.

- ▶ Non-disk-shaped object and/or pusher

- ▶ Non-disk-shaped object and/or pusher
- ▶ Finding shortest unrestricted push plans

- ▶ Non-disk-shaped object and/or pusher
- ▶ Finding shortest unrestricted push plans
- ▶ Solving the global problem (i.e. no object path given) through the configuration-space method

- ▶ Non-disk-shaped object and/or pusher
- ▶ Finding shortest unrestricted push plans
- ▶ Solving the global problem (i.e. no object path given) through the configuration-space method
- ▶ Improved running times (especially for shortest push plans)

# Questions?