Potential Field Navigation: The Distance Transform

ROB 102: Introduction to AI & Programming

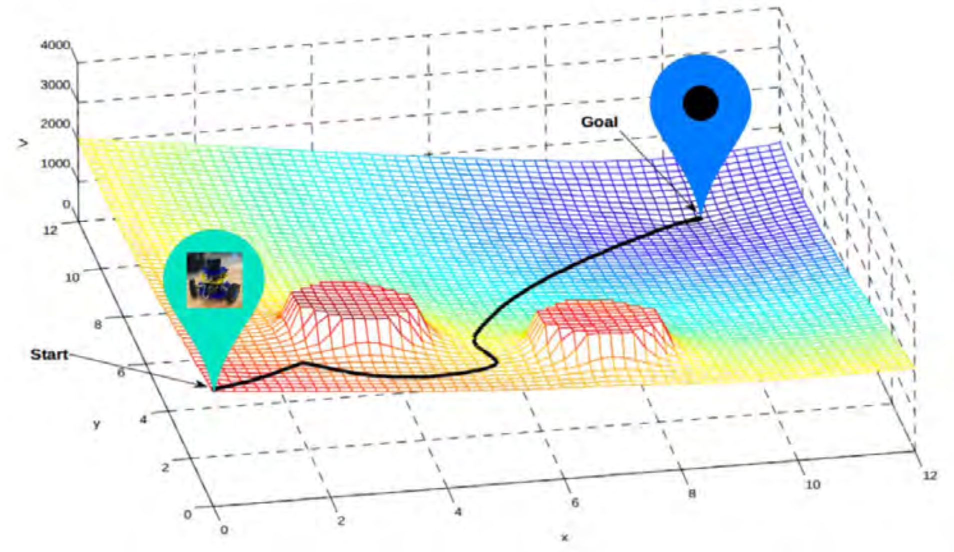
Lecture 07

2021/10/11

Project 2: Potential Field Navigation

- ❑ Build a map of environment
- ❑ Form attraction potential to goal
- ❑ Form repulsion potentials away from obstacles
- ❑ Add potentials together into potential field
- ☑ Local search over potential field to navigate

Last lecture

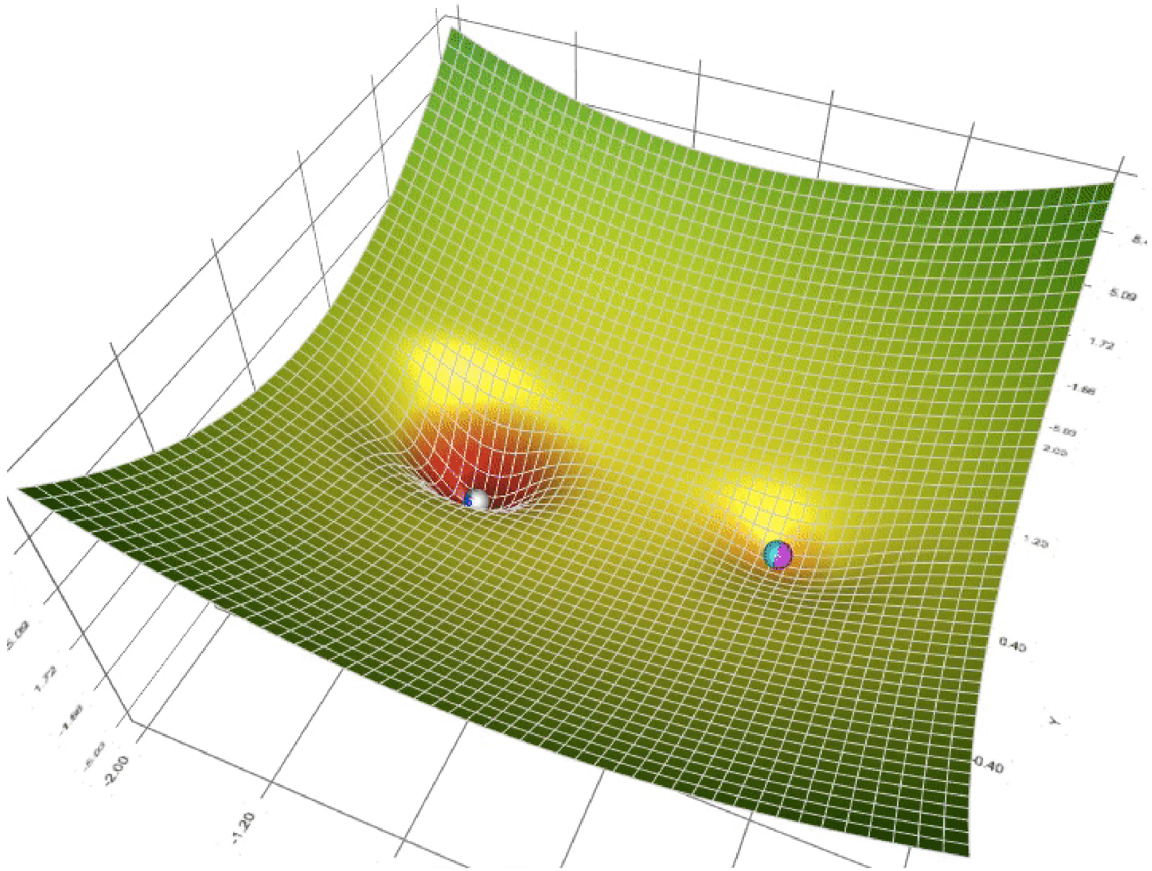


Last time…

(Link to GIF)

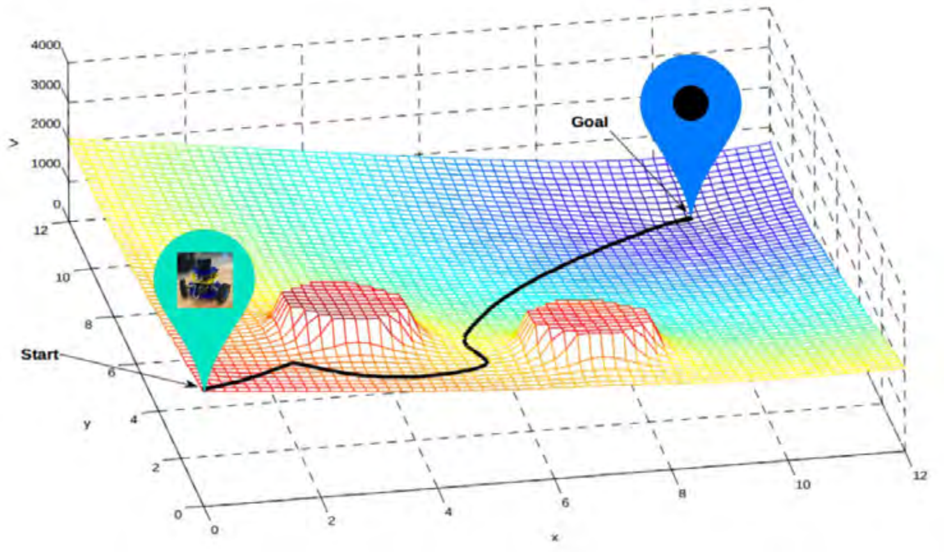
A **potential field** has *high* value in areas the robot should avoid and *low* value where the robot should go.

The robot navigates by moving to the area in its local region with the lowest potential.



Project 2: Potential Field Navigation

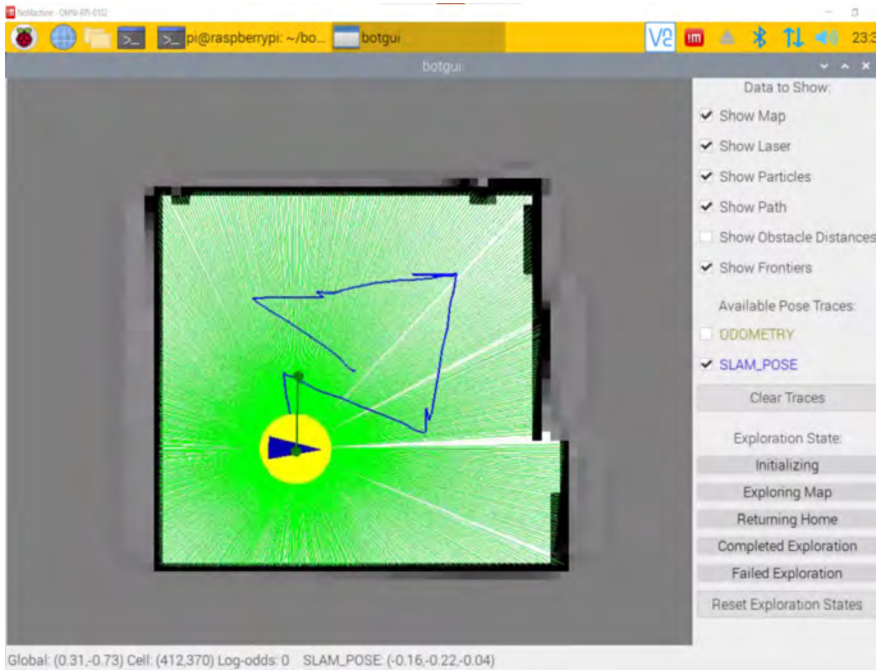
- ☑ Build a map of environment
- ☐ Form attraction potential to goal
- ☐ Form repulsion potentials away from obstacles
- ☐ Add potentials together into potential field
- ☑ Local search over potential field to navigate

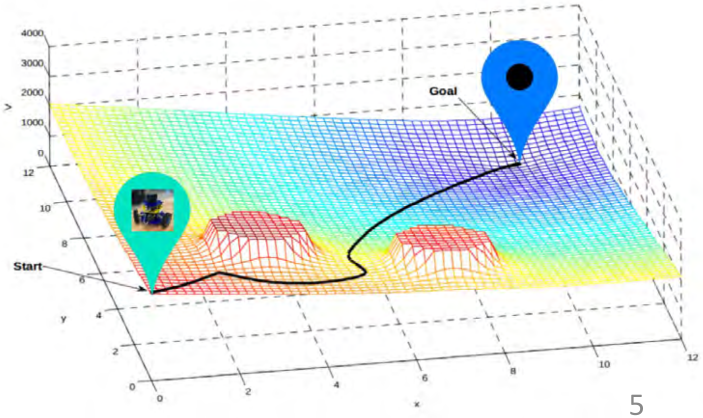


Project 2: Potential Field Navigation

- [x] Build a map of environment ← Last lab
- [ ] Form attraction potential to goal
- [ ] Form repulsion potentials away from obstacles
- [ ] Add potentials together into potential field
- [x] Local search over potential field to navigate

This lecture

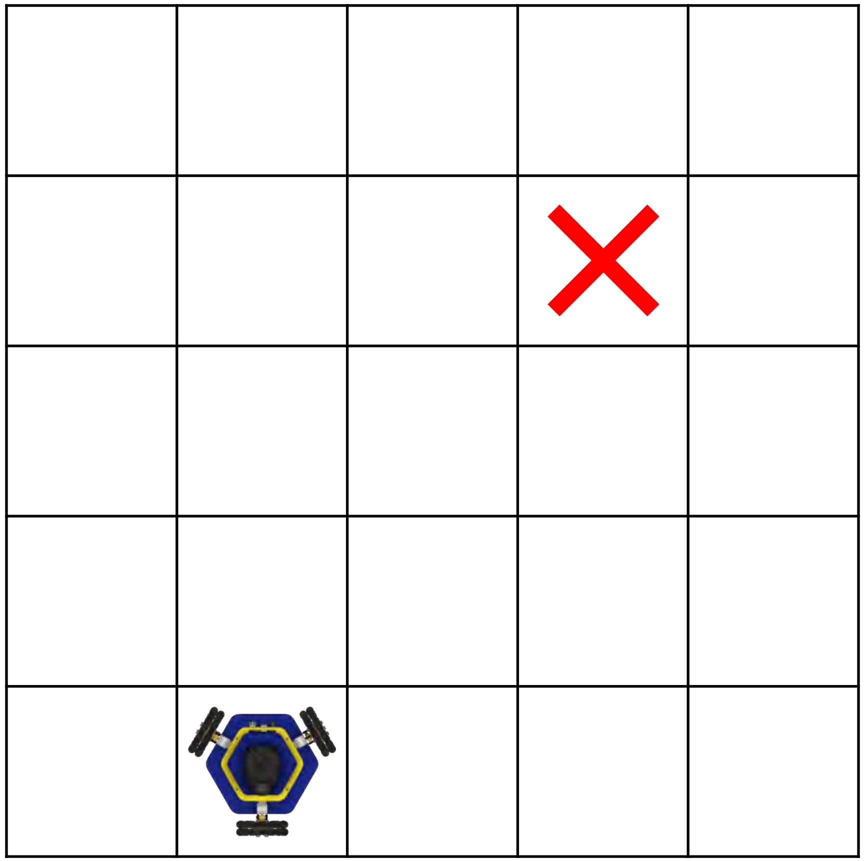




Attraction Potential

How can we make a potential that pulls the robot towards the goal?

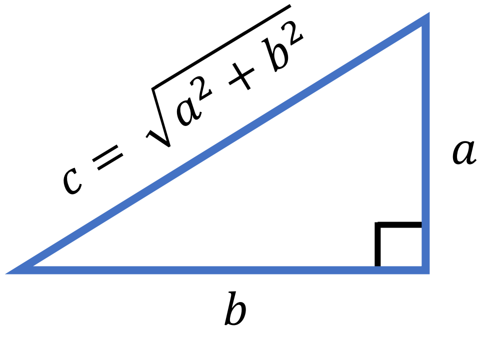
The distance from each cell to the goal makes a reasonable potential field.

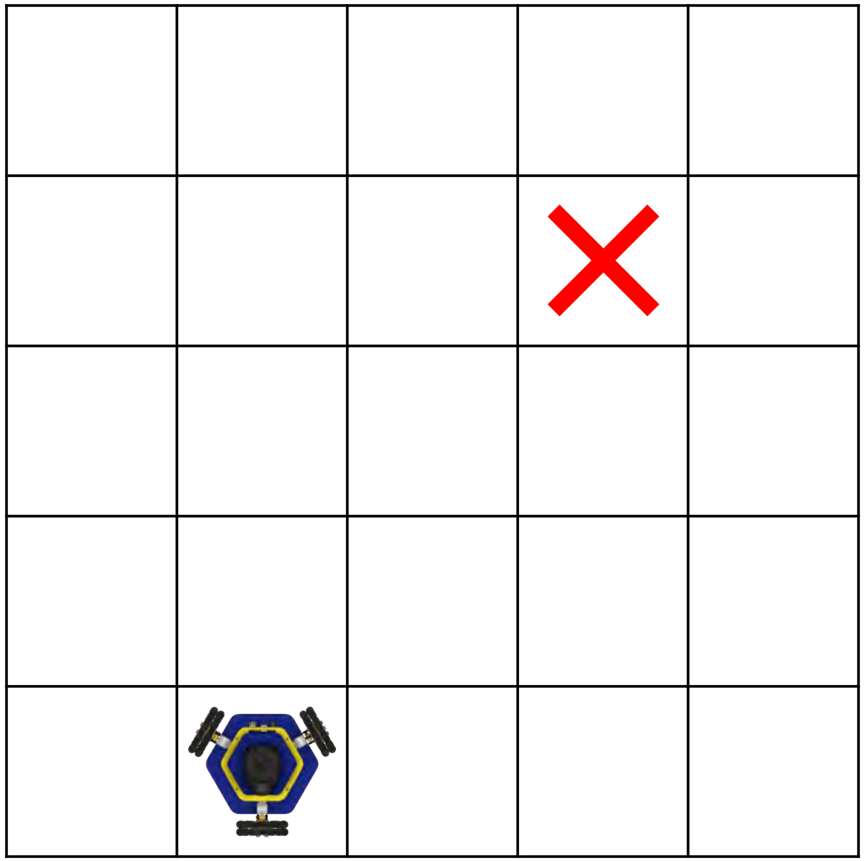


Attraction Potential

How do we define the distance between cells?

Recall: Pythagorean Rule





Recall: Storing a Map in C++

We represent the cell in terms of a coordinate in the grid.

The coordinate is written `(i, j)`, where `i` is the index of the row and `j` is the index of the column.

rows

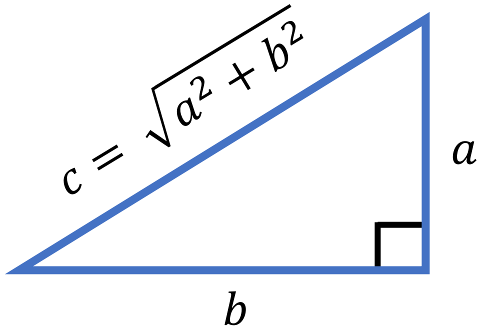
9	-128	-128	-128	127	-128	-128	-128	-128	-128	-128
8	-128	-128	-128	127	-128	-128	-128	-128	-128	-128
7	-128	-128	-128	127	-128	-128	-128	-128	-128	-128
6	-128	-128	-128	127	-128	-128	127	127	127	127
5	-128	-128	-128	127	-128	-128	-128	-128	-128	-128
4	-128	-128	-128	127	-128	-128	-128	-128	-128	-128
3	-128	-128	-128	127	-128	-128	-128	127	-128	-128
2	-128	-128	-128	-128	-128	-128	-128	127	-128	-128
1	-128	-128	-128	-128	-128	-128	-128	127	-128	-128
0	-128	-128	-128	-128	-128	-128	-128	127	-128	-128
	0	1	2	3	4	5	6	7	8	9

columns

Attraction Potential

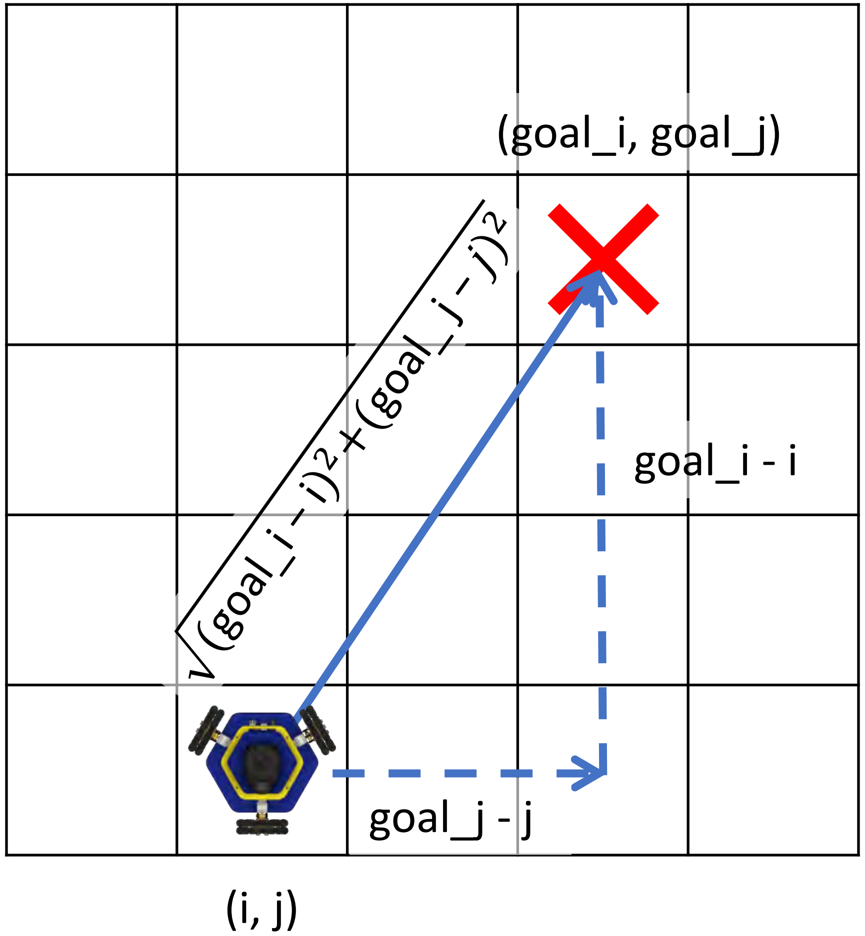
How do we define the distance between cells?

Recall: Pythagorean Rule



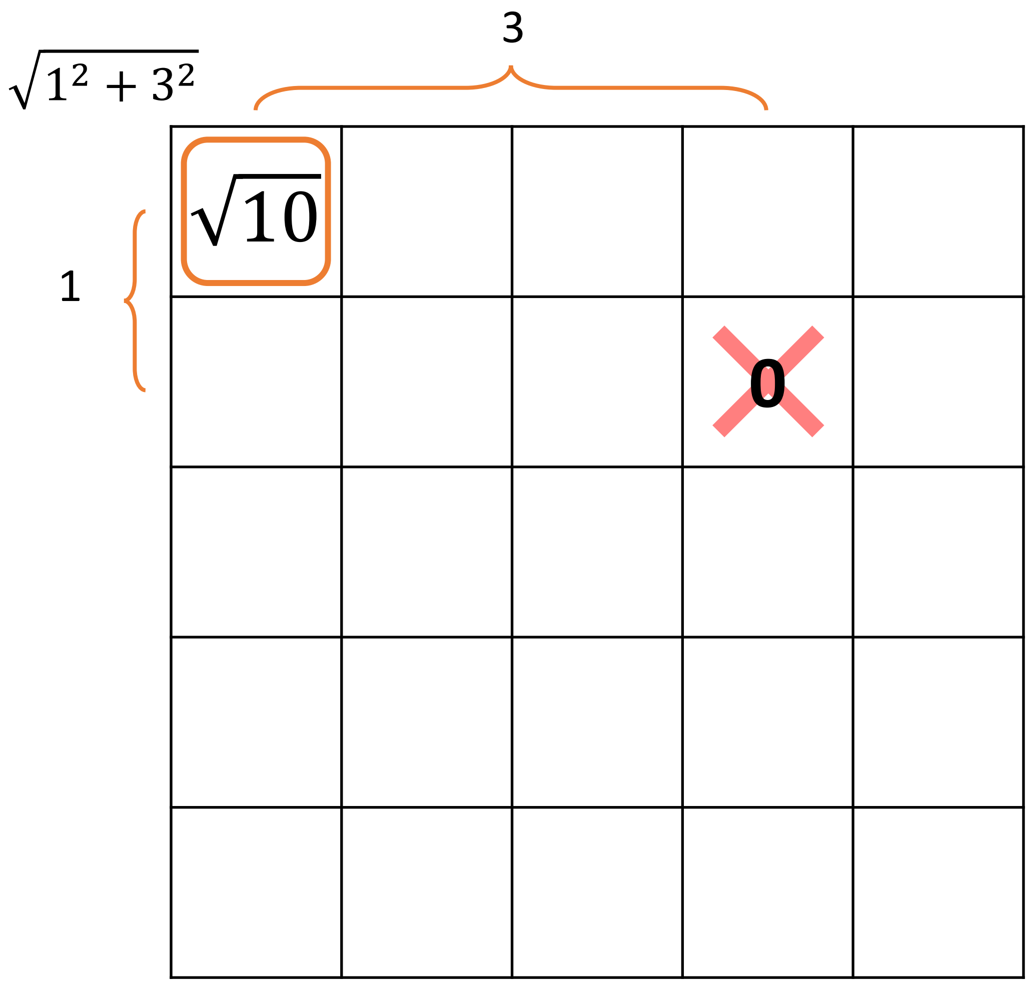
This is called the **Euclidean Distance**.

Recall: We can express a cell coordinate in terms of its row and column index: (row, col)



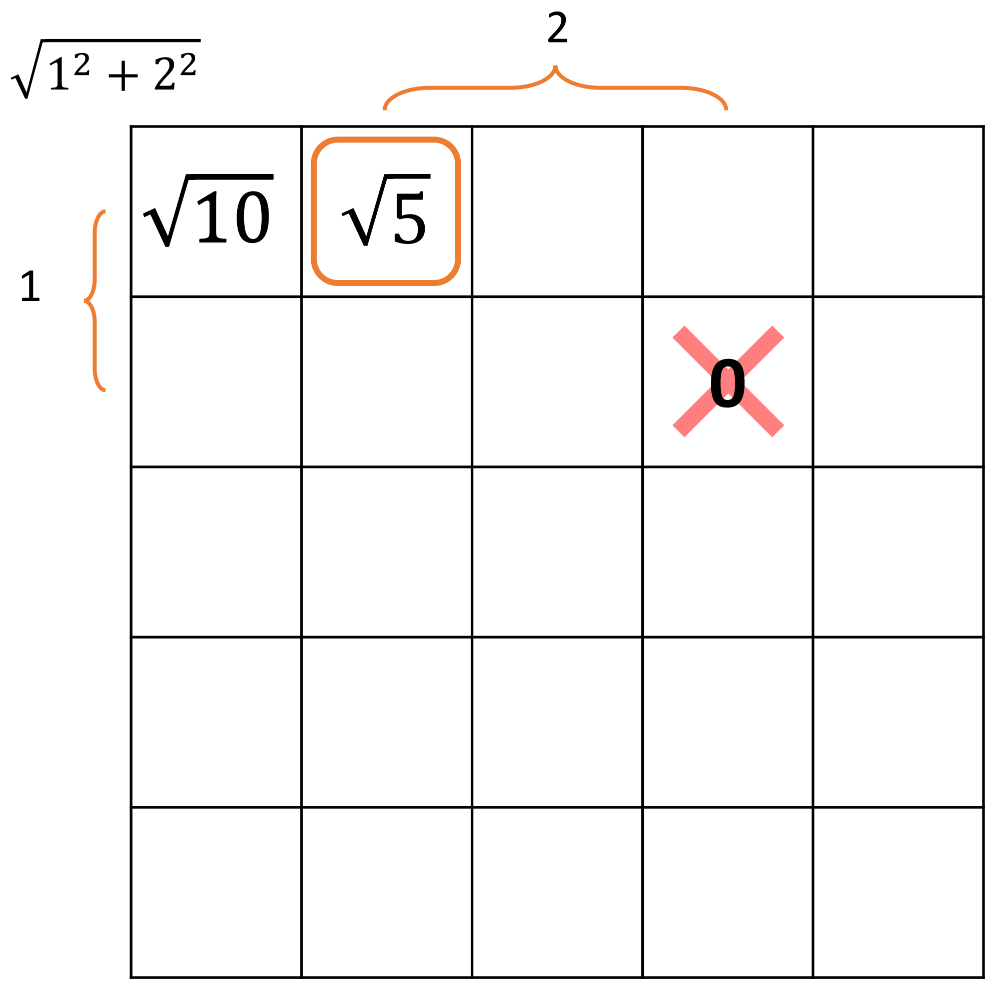
Attraction Potential

We can use the Pythagorean Rule to calculate the Euclidean distance from each cell to the goal.



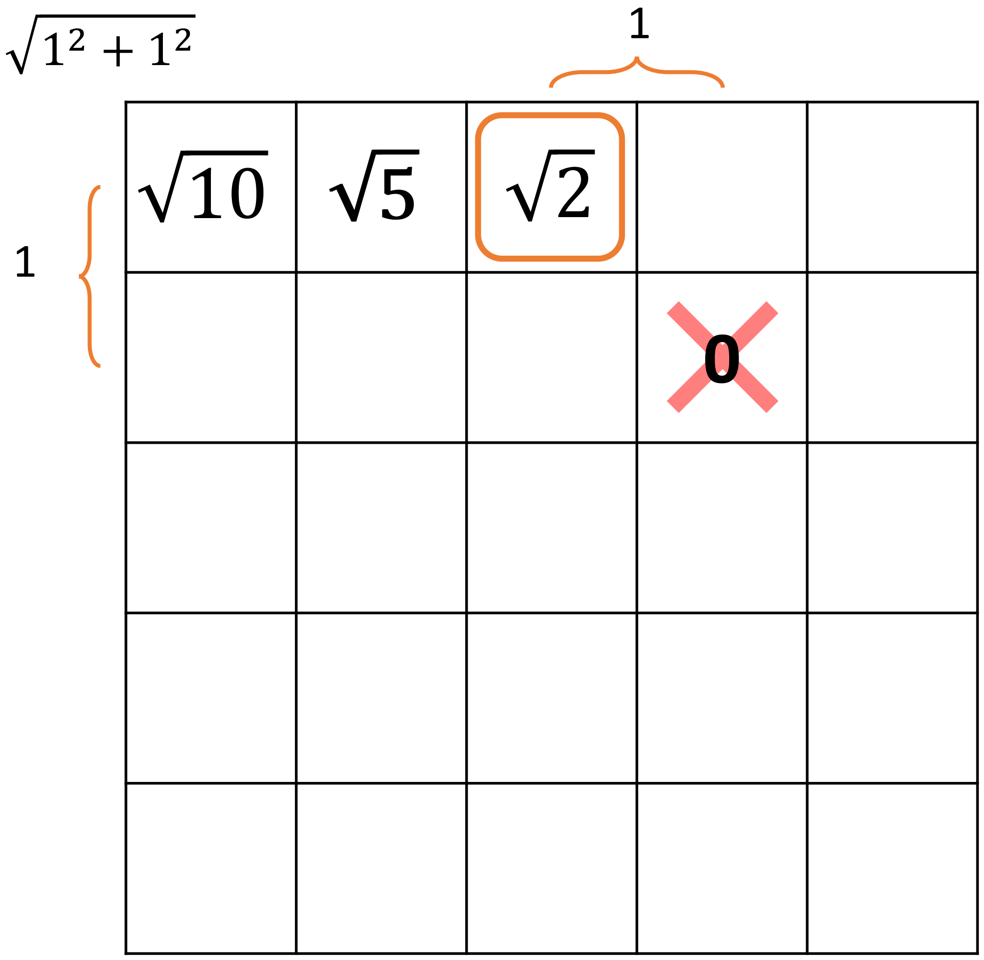
Attraction Potential

We can use the Pythagorean Rule to calculate the Euclidean distance from each cell to the goal.



Attraction Potential

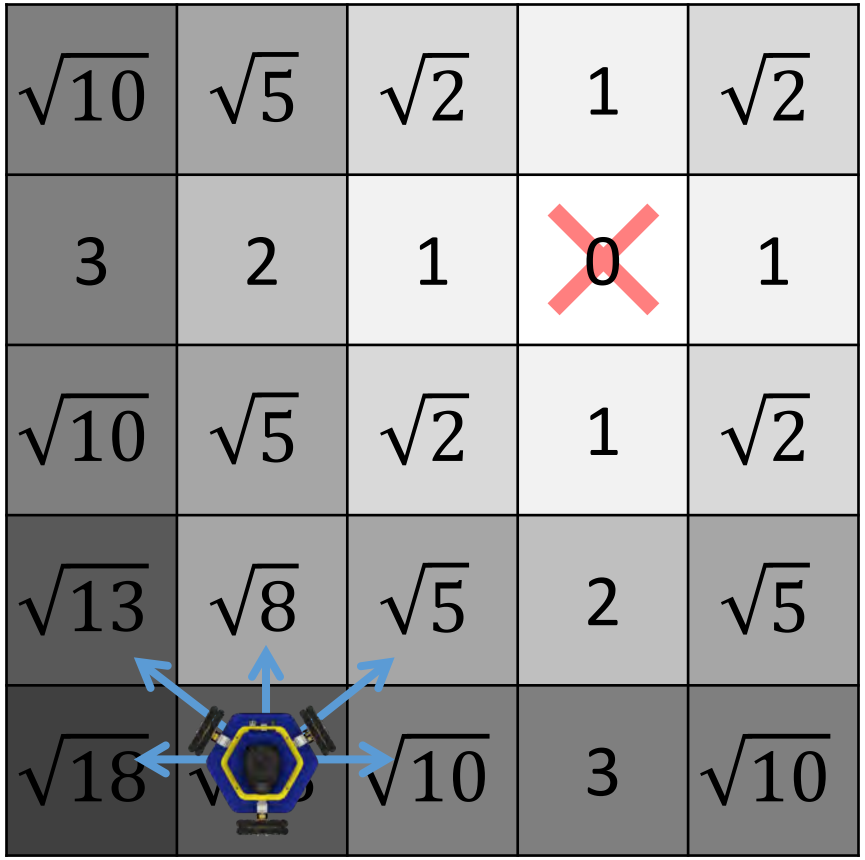
We can use the Pythagorean Rule to calculate the Euclidean distance from each cell to the goal.



Attraction Potential

We can use the Pythagorean Rule to calculate the Euclidean distance from each cell to the goal.

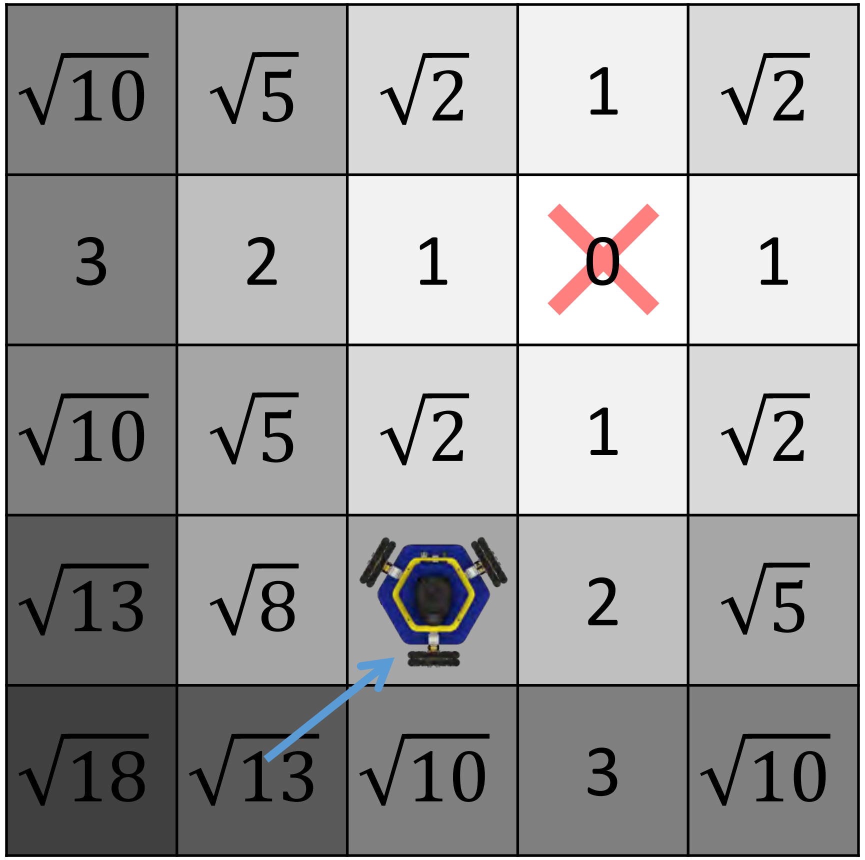
This makes a reasonable potential field which will pull the robot towards the goal.



Attraction Potential

We can use the Pythagorean Rule to calculate the Euclidean distance from each cell to the goal.

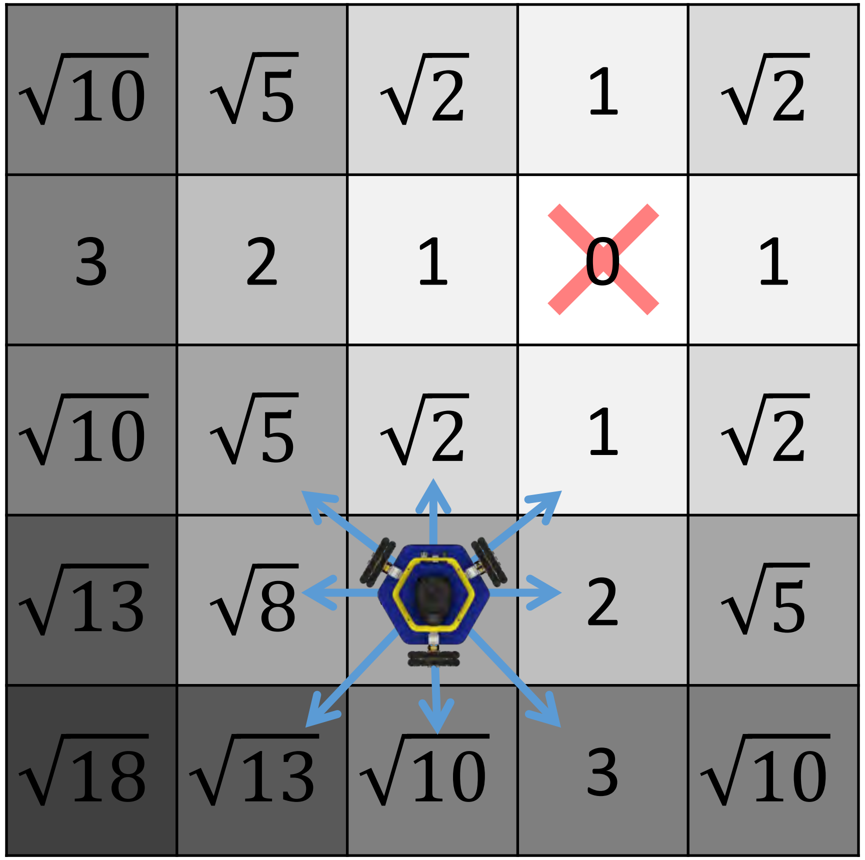
This makes a reasonable potential field which will pull the robot towards the goal.



Attraction Potential

We can use the Pythagorean Rule to calculate the Euclidean distance from each cell to the goal.

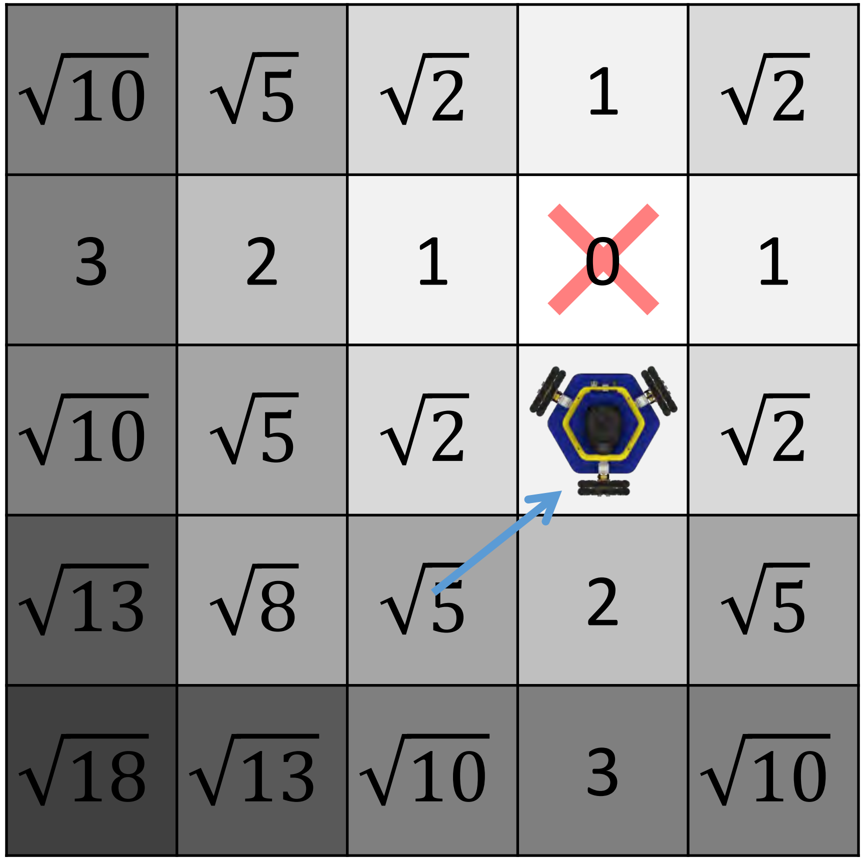
This makes a reasonable potential field which will pull the robot towards the goal.



Attraction Potential

We can use the Pythagorean Rule to calculate the Euclidean distance from each cell to the goal.

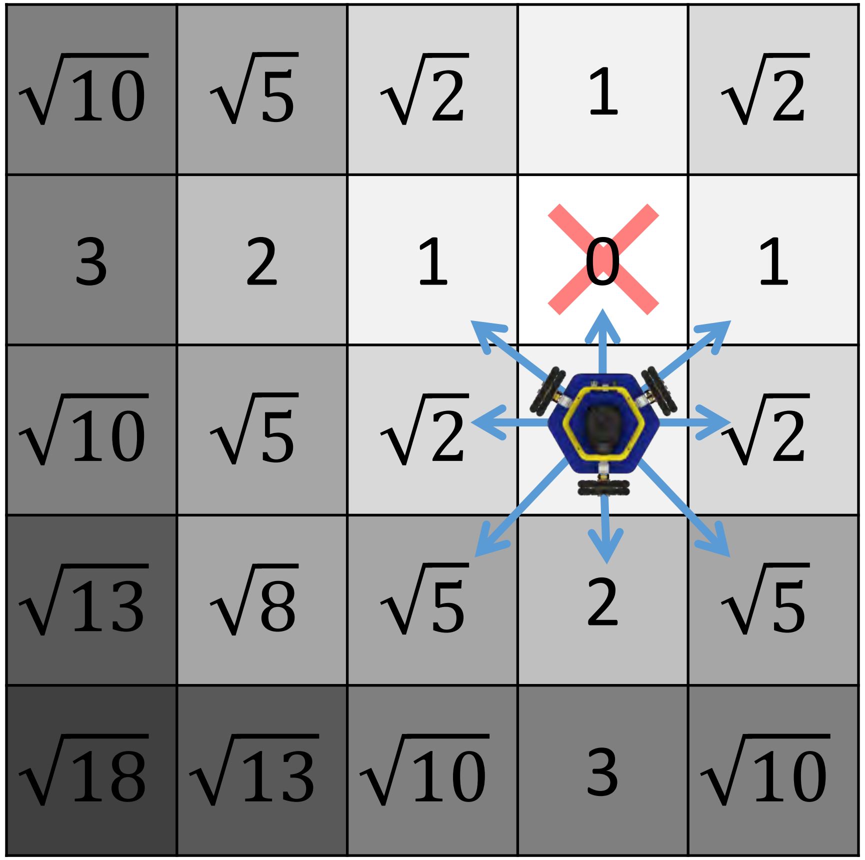
This makes a reasonable potential field which will pull the robot towards the goal.



Attraction Potential

We can use the Pythagorean Rule to calculate the Euclidean distance from each cell to the goal.

This makes a reasonable potential field which will pull the robot towards the goal.



Attraction Potential

We can use the Pythagorean Rule to calculate the Euclidean distance from each cell to the goal.

This makes a reasonable potential field which will pull the robot towards the goal.

$\sqrt{10}$	$\sqrt{5}$	$\sqrt{2}$	1	$\sqrt{2}$
3	2	1		1
$\sqrt{10}$	$\sqrt{5}$	$\sqrt{2}$	1	$\sqrt{2}$
$\sqrt{13}$	$\sqrt{8}$	$\sqrt{5}$	2	$\sqrt{5}$
$\sqrt{18}$	$\sqrt{13}$	$\sqrt{10}$	3	$\sqrt{10}$

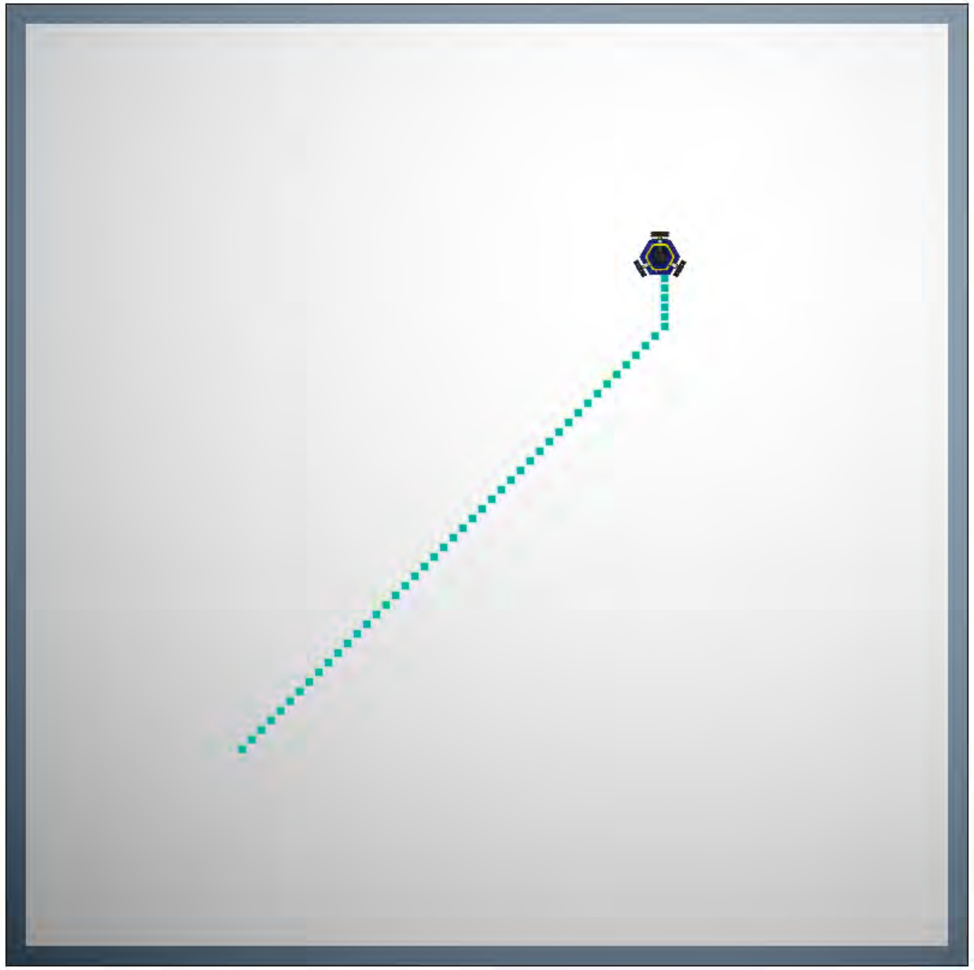
Example: Attraction Potential

That works!

You will do this in P2.1.

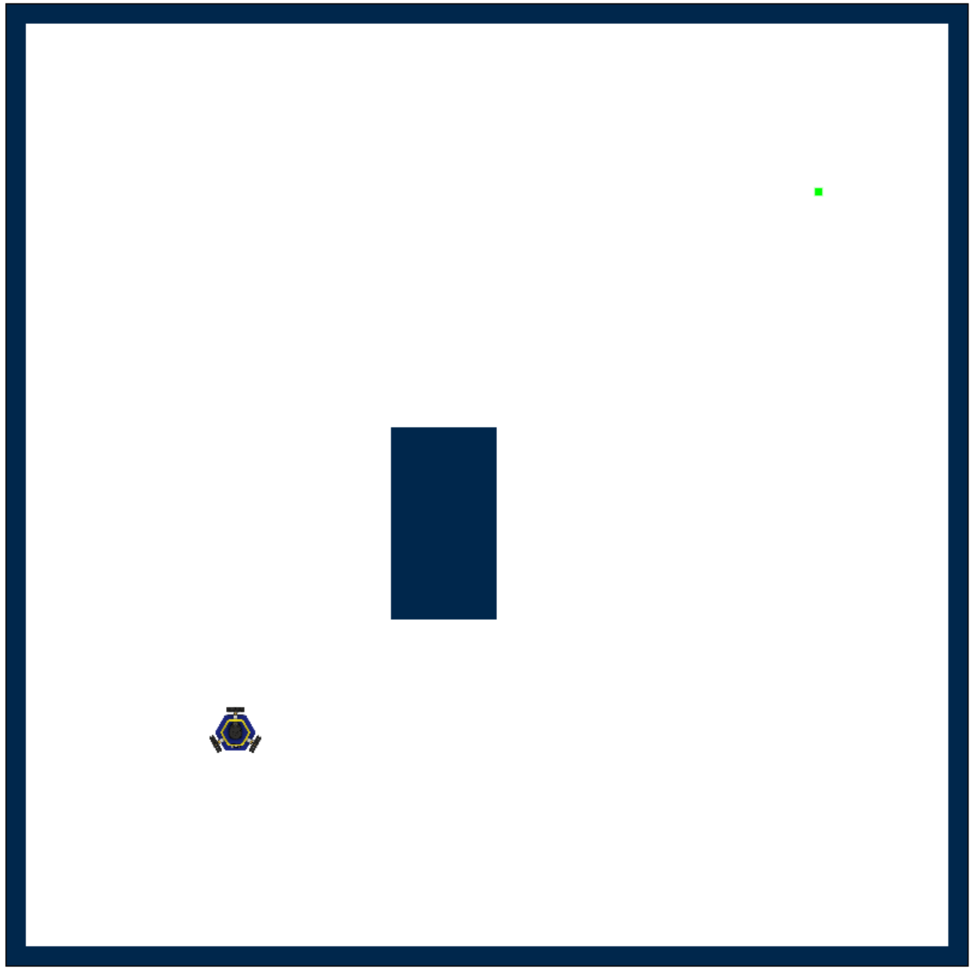
```
std::vector<float> createAttractiveField(GridGraph& graph, const Cell& goal)
{
    std::vector<float> attractive_field(graph.width * graph.height, HIGH);

    /**
     * TODO (P2): Using the graph and the given goal, create an attractive field
     * which pulls the robot towards the goal. It should be HIGH when far away
     * from the goal, and LOW when close to the goal.
     *
     * Store the result in the vector attractive_field, which should be indexed
     * the same way as the graph cell data.
     **/

    return attractive_field;
}
```

`src/potential_field/potential_field.cpp`



Dealing with Obstacles

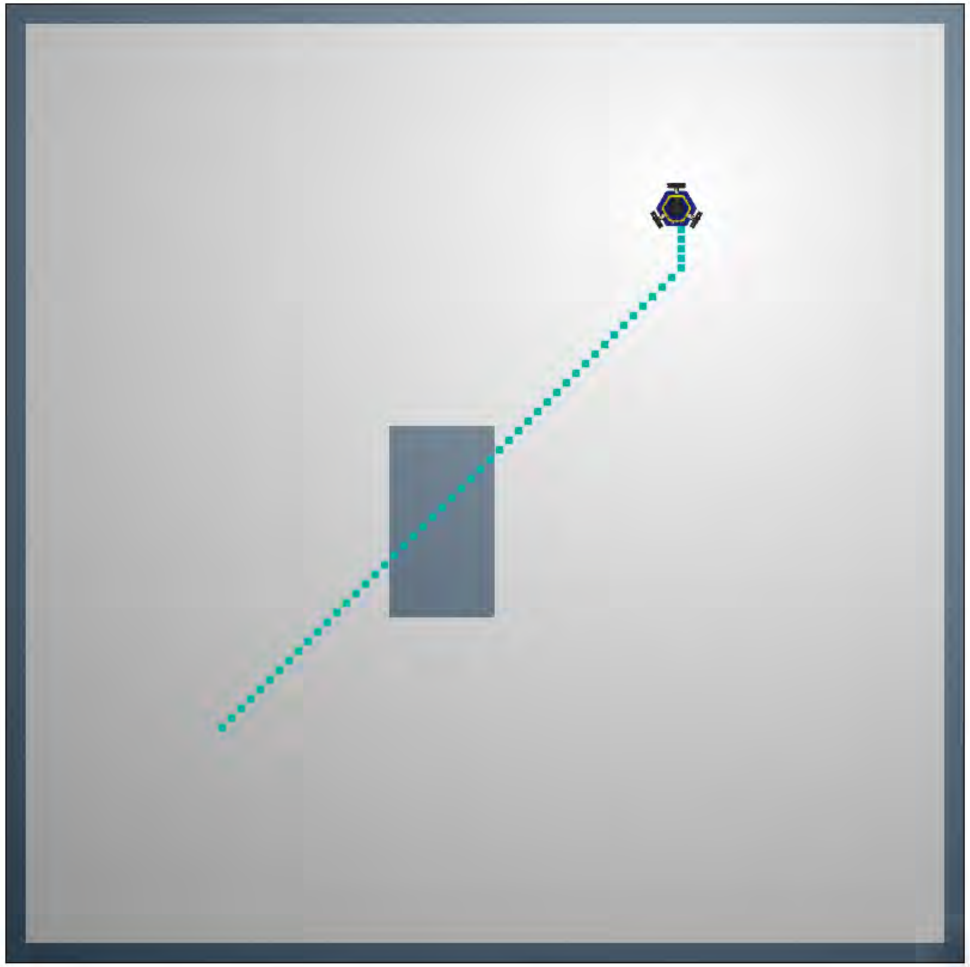
What happens if there is an obstacle in the way?



Dealing with Obstacles

With just an attraction potential, the robot will try to go right through obstacles!

We need a way to repel the robot away from obstacles.

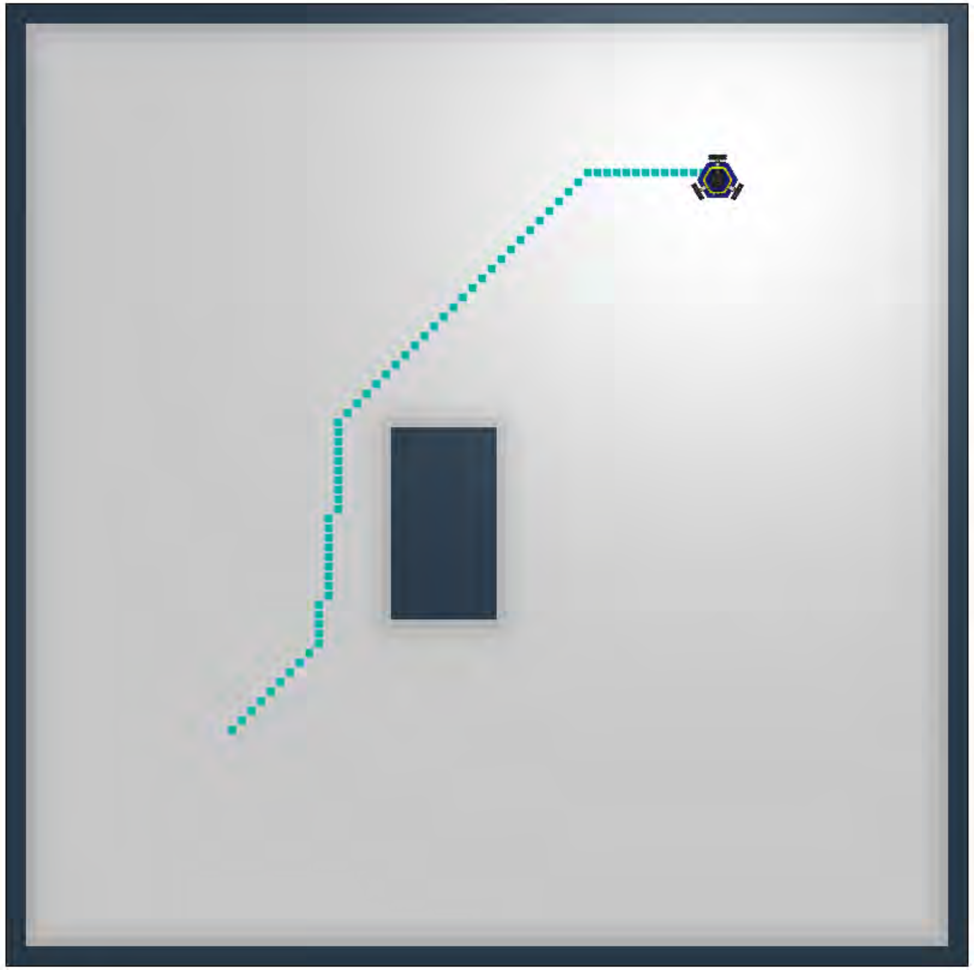


The Repulsion Potential

We can add another potential that pushes the robot away from obstacles.

Our final potential field will be a combination of the attraction and repulsion potentials.

Next lecture: How to combine potentials.

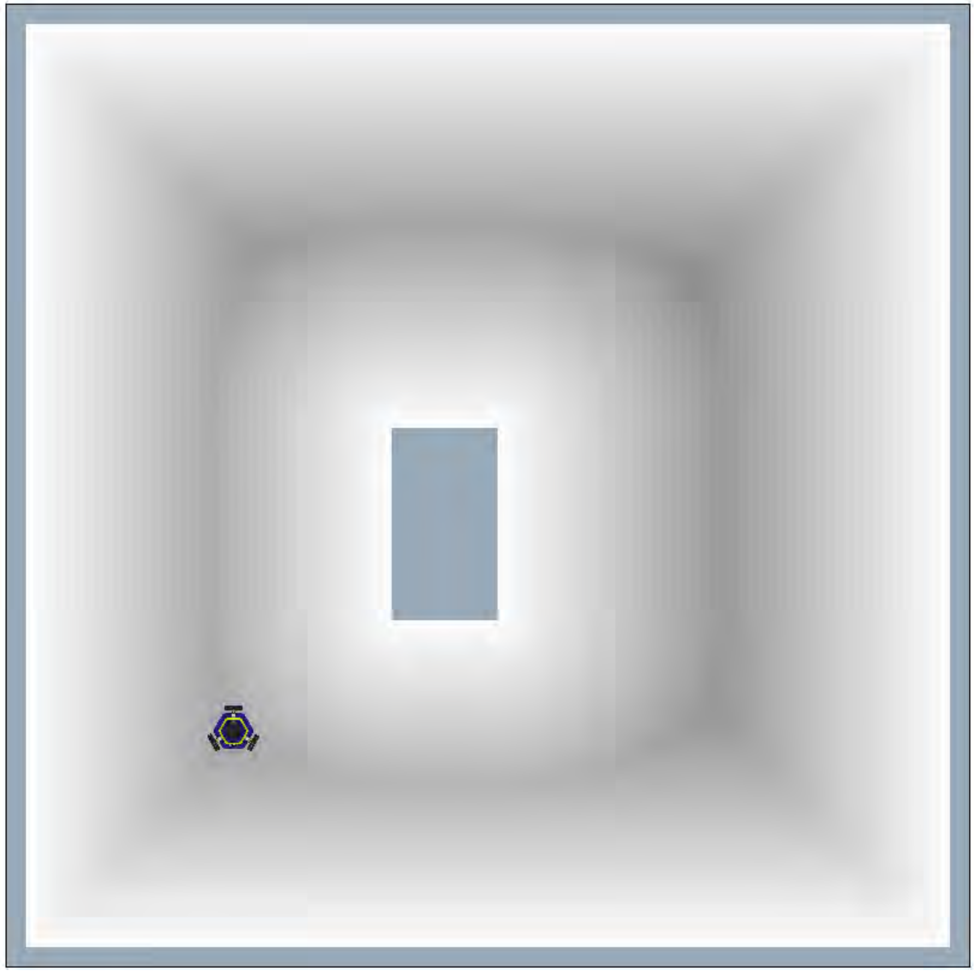


The Distance Transform

The **distance transform** is an algorithm that calculates the distance from each cell to the nearest occupied cell.

We will see two algorithms to compute the distance transform.

Next lecture: How to turn a distance transform into a repulsive field.

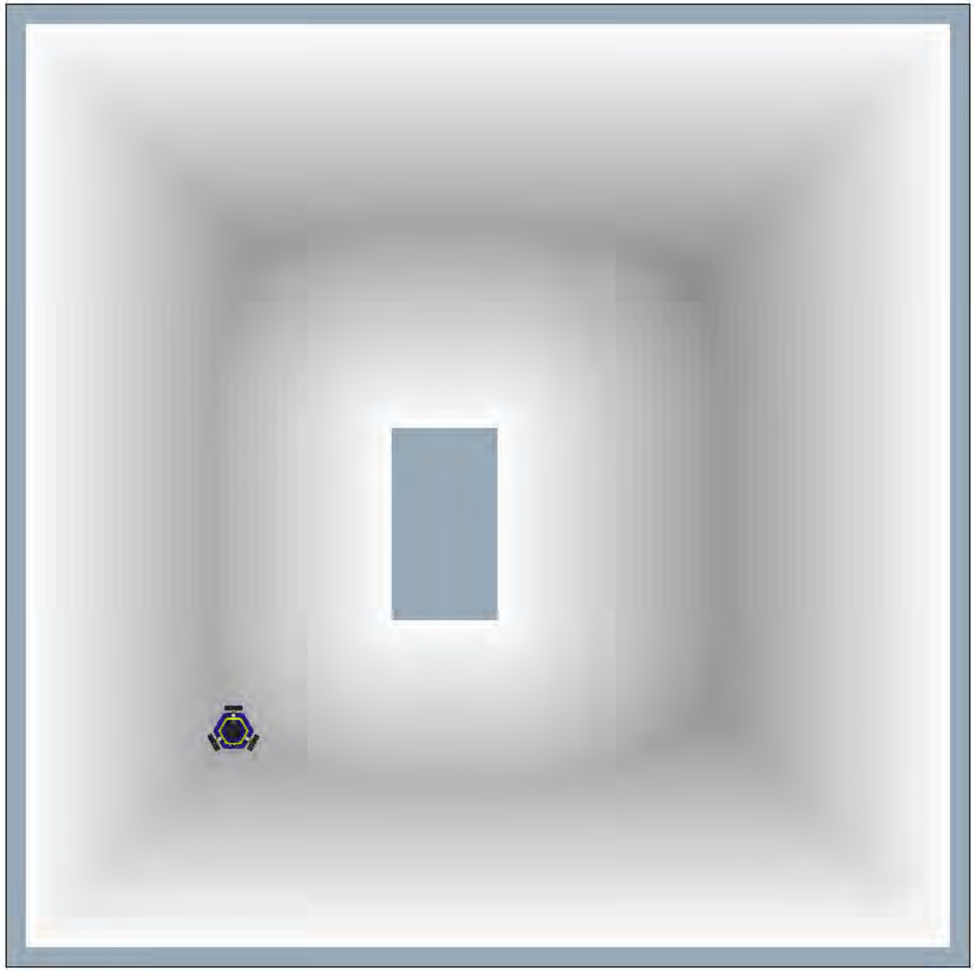


The Distance Transform

How do we calculate a distance transform?

Idea: For each cell, we could check the distance to every occupied cell in the graph.

Recall: We need to decide what “distance” means. We’ll use the Euclidean distance.



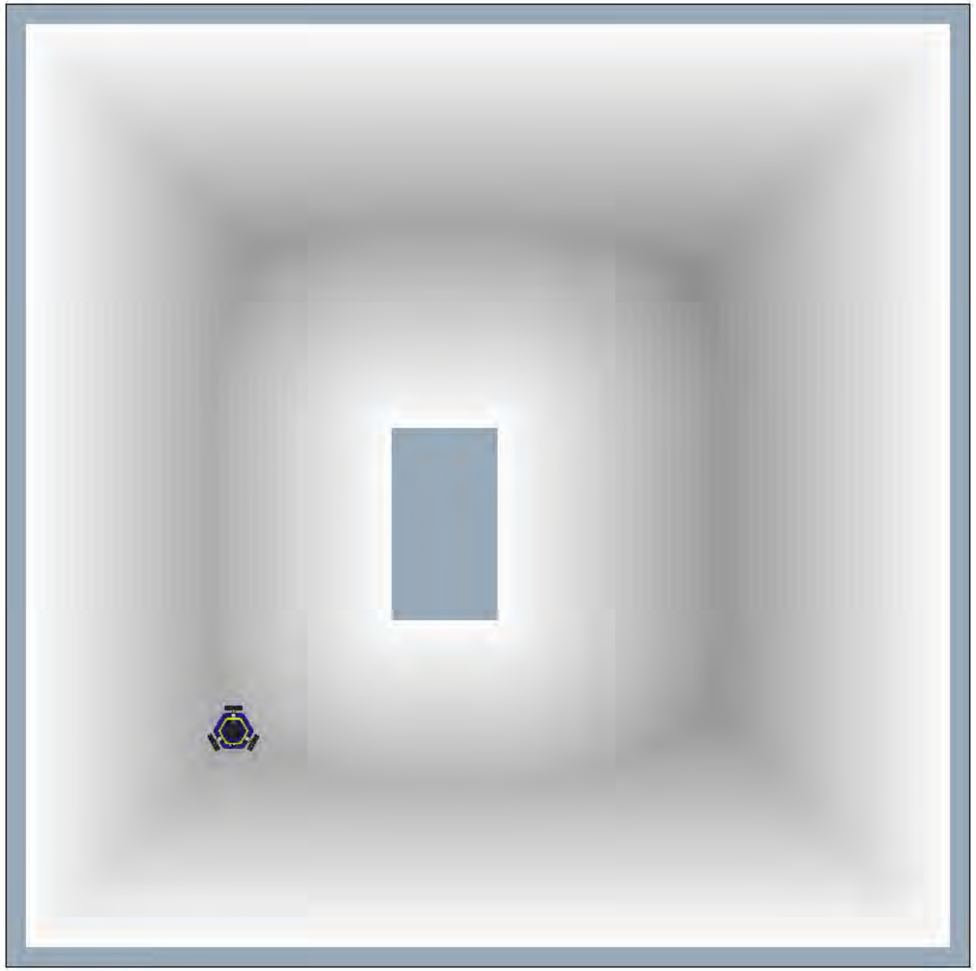
Computing the Distance Transform

Given a graph with width W and height H, with N `= W*H` cells:

$\boxed{\texttt{dist\_tf} \leftarrow \text{Vector of length N}}$

Initialize the distance transform vector

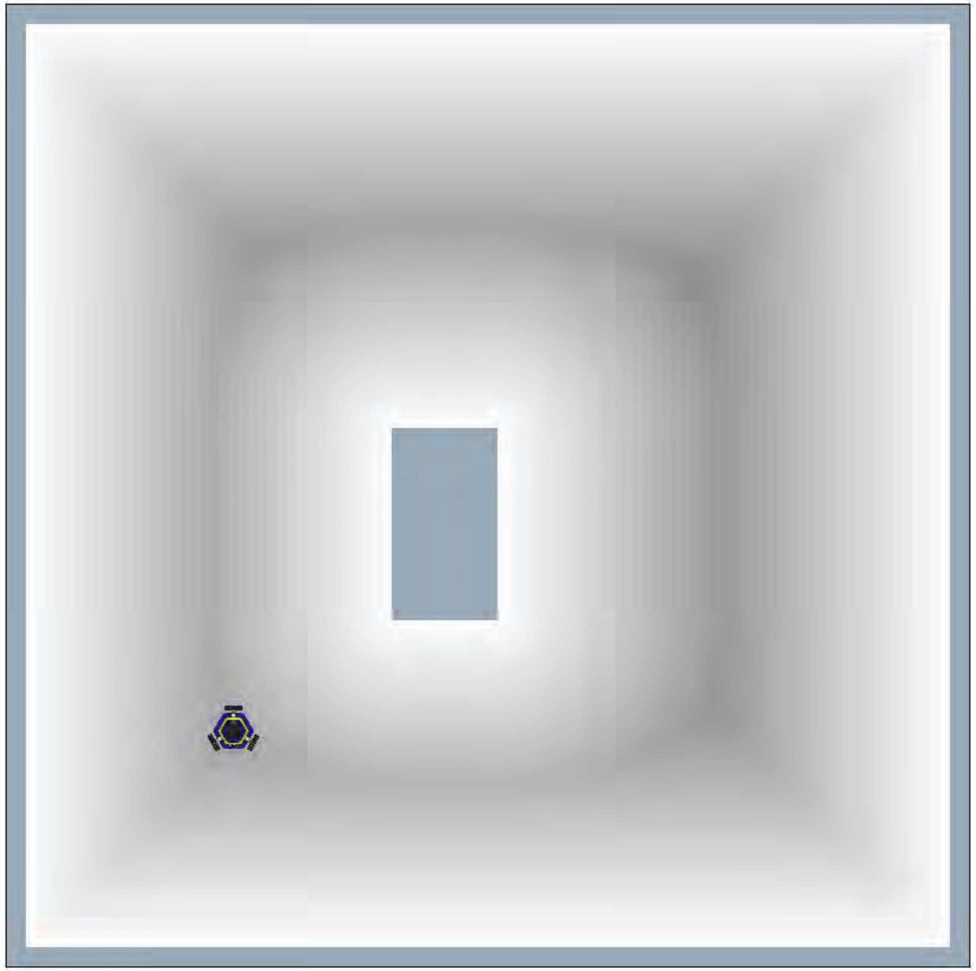
```
for i=0 to N-1 do:
  min_dist = HIGH
  for j=0 to N-1 do:
    if graph[j] is occupied:
      dist ← Euclidean distance from cell i to j
      if dist < min_dist:
        min_dist = dist
  dist_tf[i] = min_dist
```



Computing the Distance Transform

Given a graph with width W and height H, with `N = W*H` cells:

```
dist_tf ← Vector of length N          Initialize
for i=0 to N-1 do:                    Loop through every cell
  min_dist = HIGH
  for j=0 to N-1 do:
    if graph[j] is occupied:
      dist ← Euclidean distance from cell i to j
      if dist < min_dist:
        min_dist = dist
  dist_tf[i] = min_dist
```

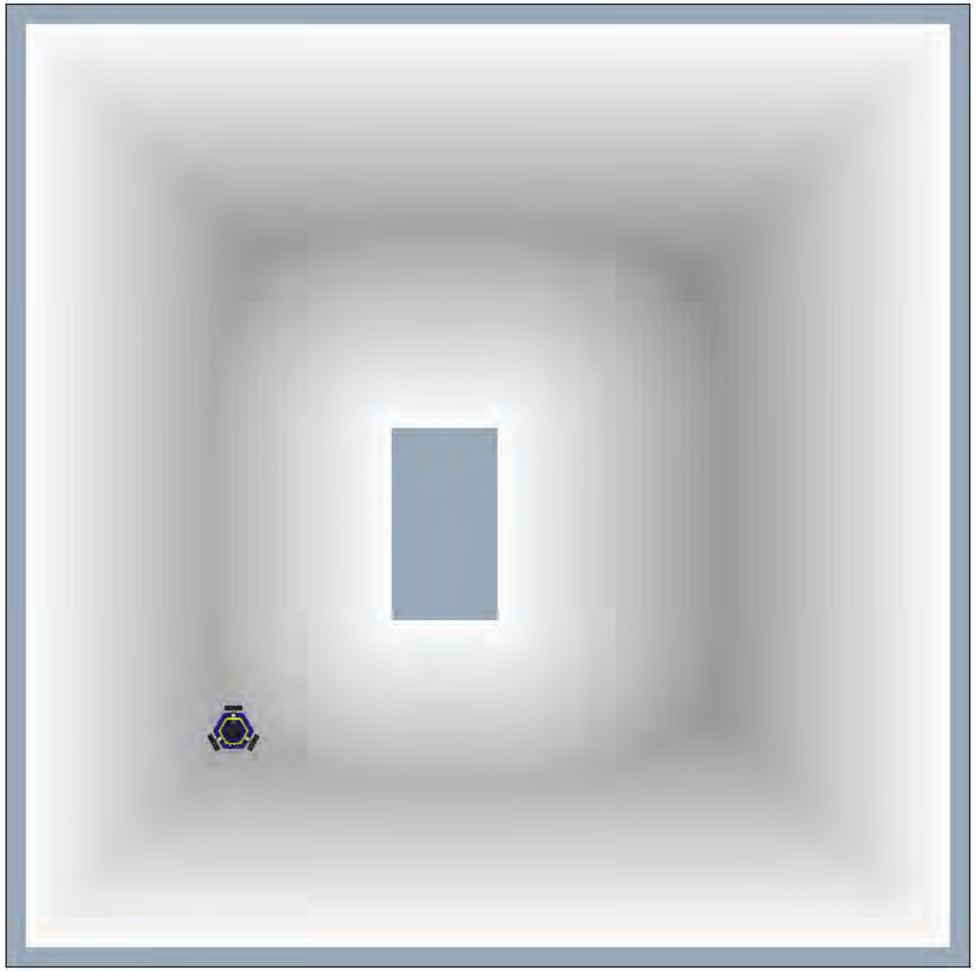


Computing the Distance Transform

Given a graph with width W and height H, with N = W*H cells:

dist_tf ← Vector of length N Initialize

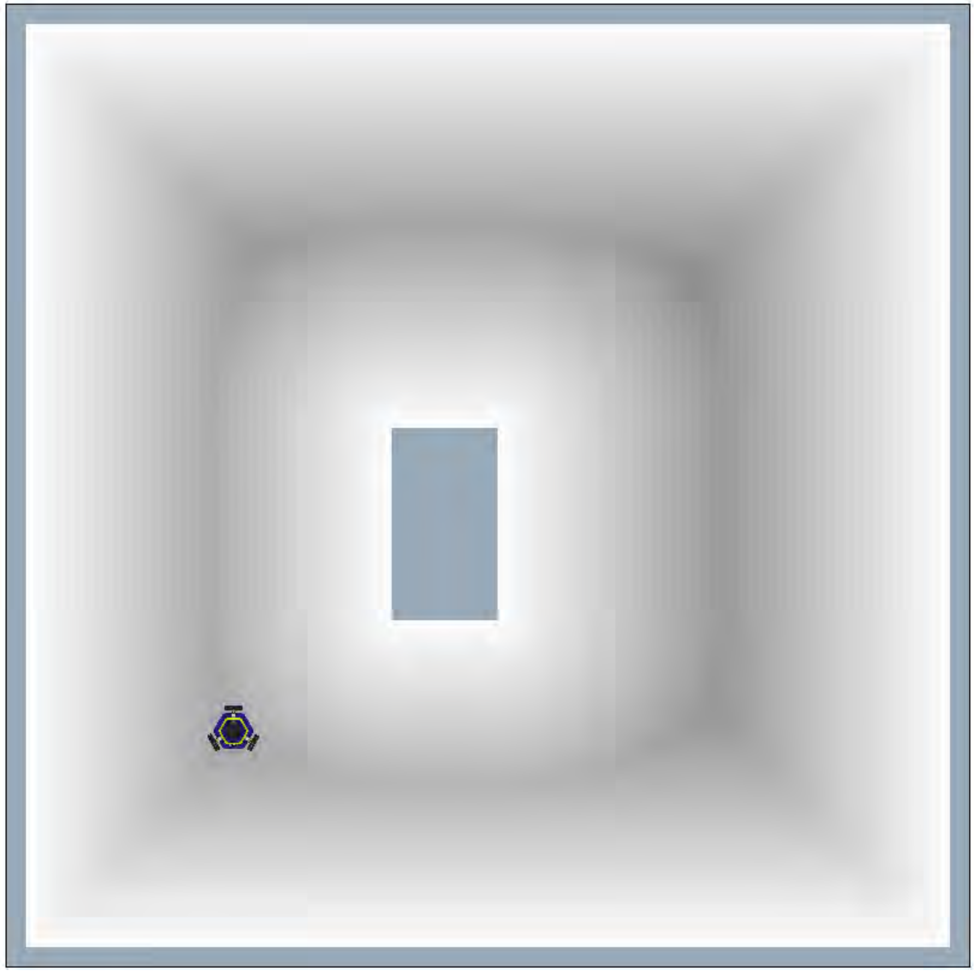
```
for i=0 to N-1 do:                Loop through every cell
  min_dist = HIGH                 Initialize the minimum distance
  for j=0 to N-1 do:
    if graph[j] is occupied:
      dist ← Euclidean distance from cell i to j
      if dist < min_dist:
        min_dist = dist
  dist_tf[i] = min_dist
```



Computing the Distance Transform

Given a graph with width W and height H, with N = W*H cells:

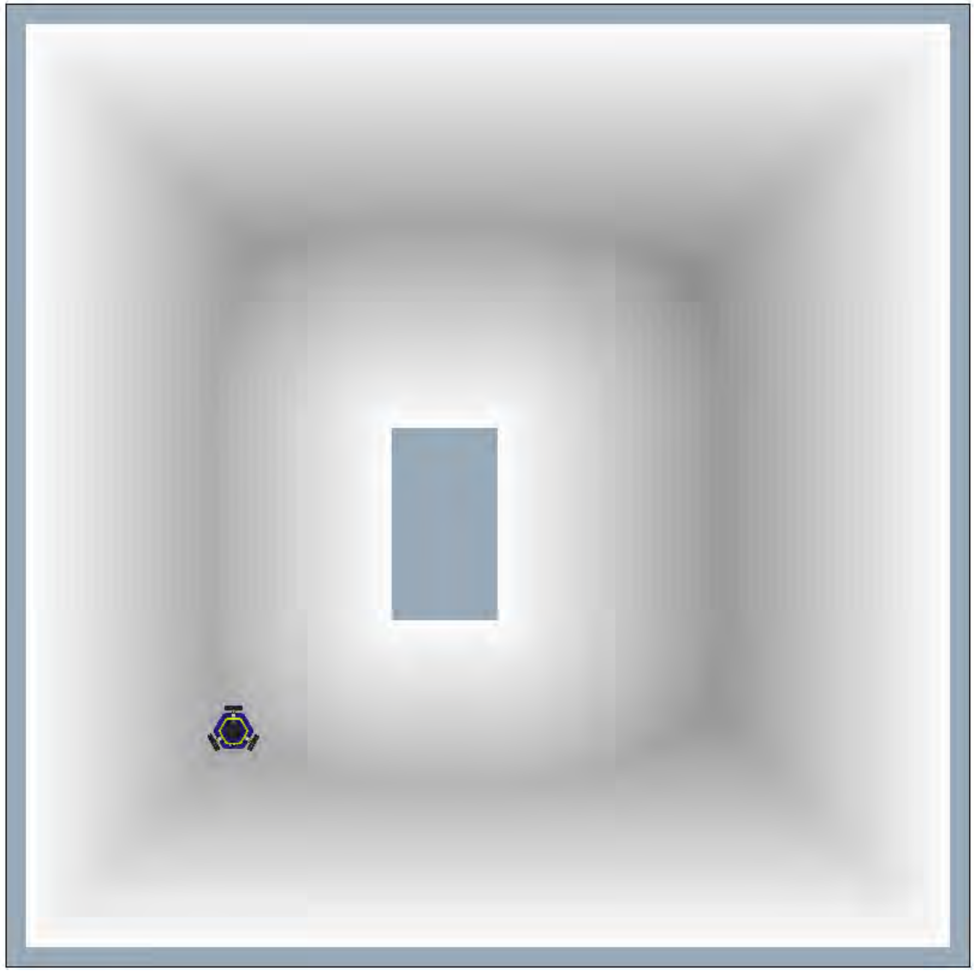
```
dist_tf ← Vector of length N          Initialize
for i=0 to N-1 do:                    Loop through every cell
  min_dist = HIGH                     Initialize the minimum distance
  for j=0 to N-1 do:                  Loop through every cell
    if graph[j] is occupied:
      dist ← Euclidean distance from cell i to j
      if dist < min_dist:
        min_dist = dist
  dist_tf[i] = min_dist
```



Computing the Distance Transform

Given a graph with width W and height H, with N = W*H cells:

```
dist_tf ← Vector of length N                Initialize
for i=0 to N-1 do:                          Loop through every cell
  min_dist = HIGH                           Initialize the minimum distance
  for j=0 to N-1 do:                        Loop through every cell
    if graph[j] is occupied:
      dist ← Euclidean distance from cell i to j
      if dist < min_dist:                   Keep track of the closest
        min_dist = dist                     occupied cell to i
  dist_tf[i] = min_dist
```

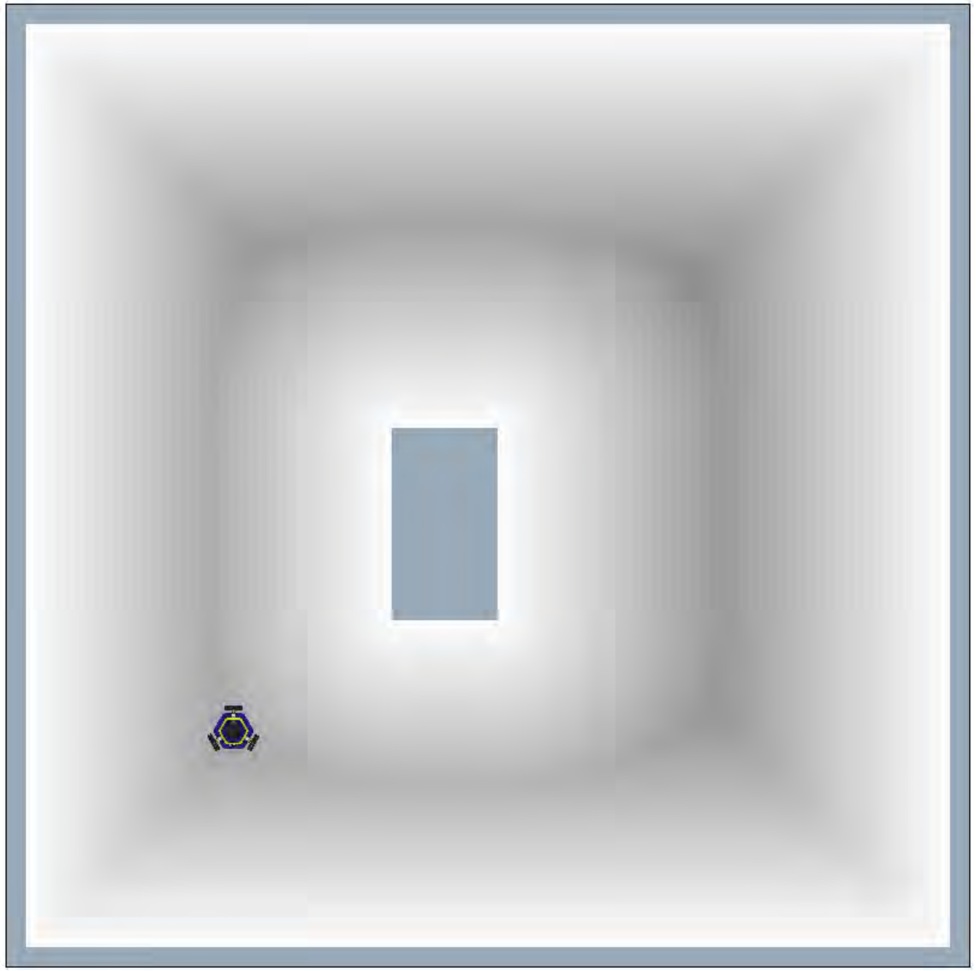


Computing the Distance Transform

Given a graph with width W and height H,
with N `= W*H` cells:

```
dist_tf ← Vector of length N                  Initialize
for i=0 to N-1 do:                            Loop through every cell
  min_dist = HIGH                             Initialize the minimum distance
  for j=0 to N-1 do:                          Loop through every cell
    if graph[j] is occupied:
      dist ← Euclidean distance from cell i to j
      if dist < min_dist:                     Keep track of the closest
        min_dist = dist                       occupied cell to i
  dist_tf[i] = min_dist
```

Update the distance transform

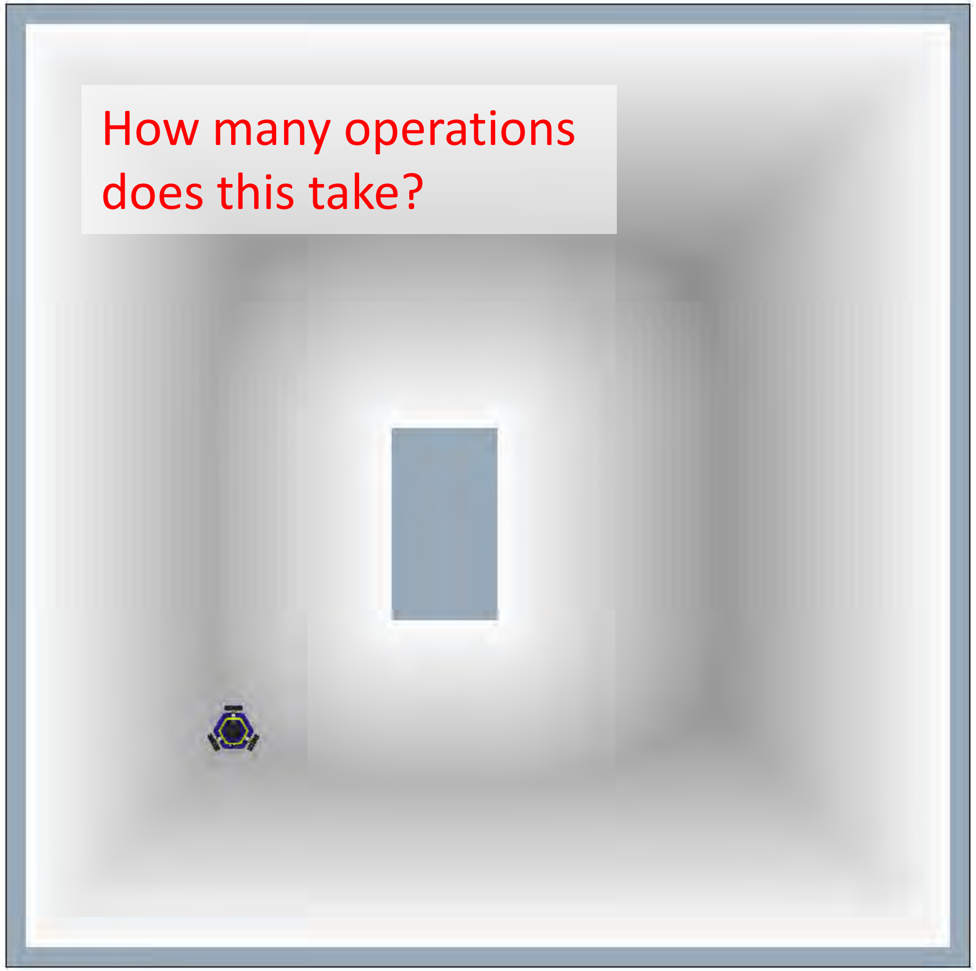


Computing the Distance Transform

Given a graph with width W and height H, with N = W*H cells:

```
dist_tf ← Vector of length N                      Initialize
for i=0 to N-1 do:                                 Loop through every cell
  min_dist = HIGH                                  Initialize the minimum distance
  for j=0 to N-1 do:                               Loop through every cell
    if graph[j] is occupied:
      dist ← Euclidean distance from cell i to j
      if dist < min_dist:                          Keep track of the closest
        min_dist = dist                            occupied cell to i
  dist_tf[i] = min_dist
```

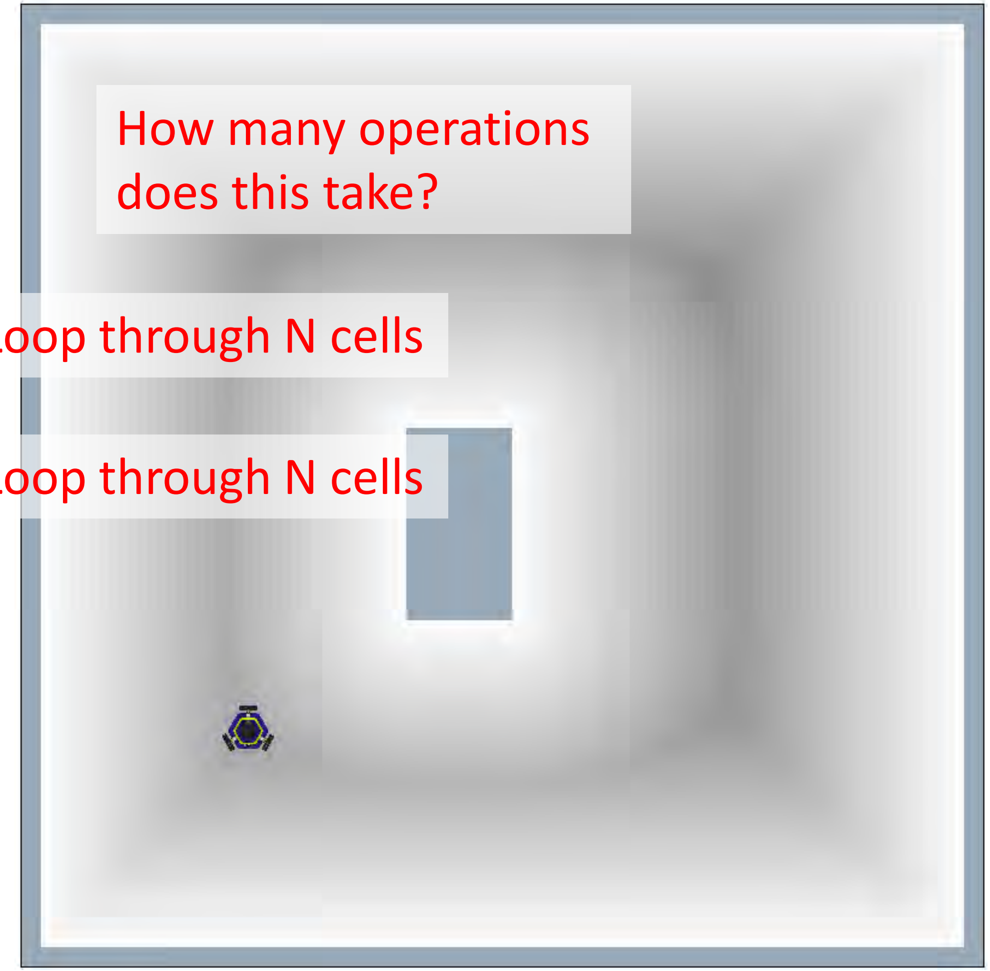
Update the distance transform



Computing the Distance Transform

Given a graph with width W and height H, with N = W*H cells:

```
dist_tf ← Vector of length N
for i=0 to N-1 do:      Loop through every cell
  min_dist = HIGH
  for j=0 to N-1 do:      Loop through every cell
    if graph[j] is occupied:
      dist ← Euclidean distance from cell i to j
      if dist < min_dist:
        min_dist = dist
  dist_tf[i] = min_dist
```

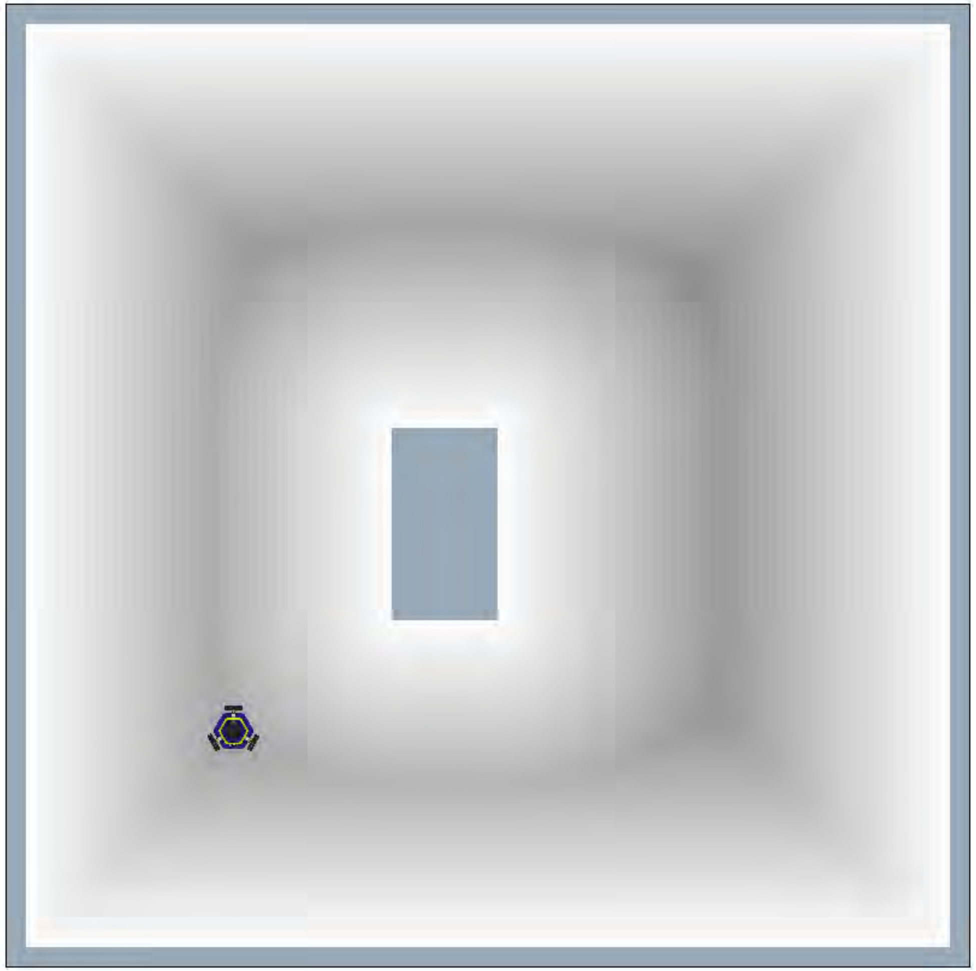


Computing the Distance Transform

How many operations does this take?

We did N loops N times, or N^2 loops total.

Remember, N is the number of cells in the graph. As the graph gets bigger, this gets very slow!

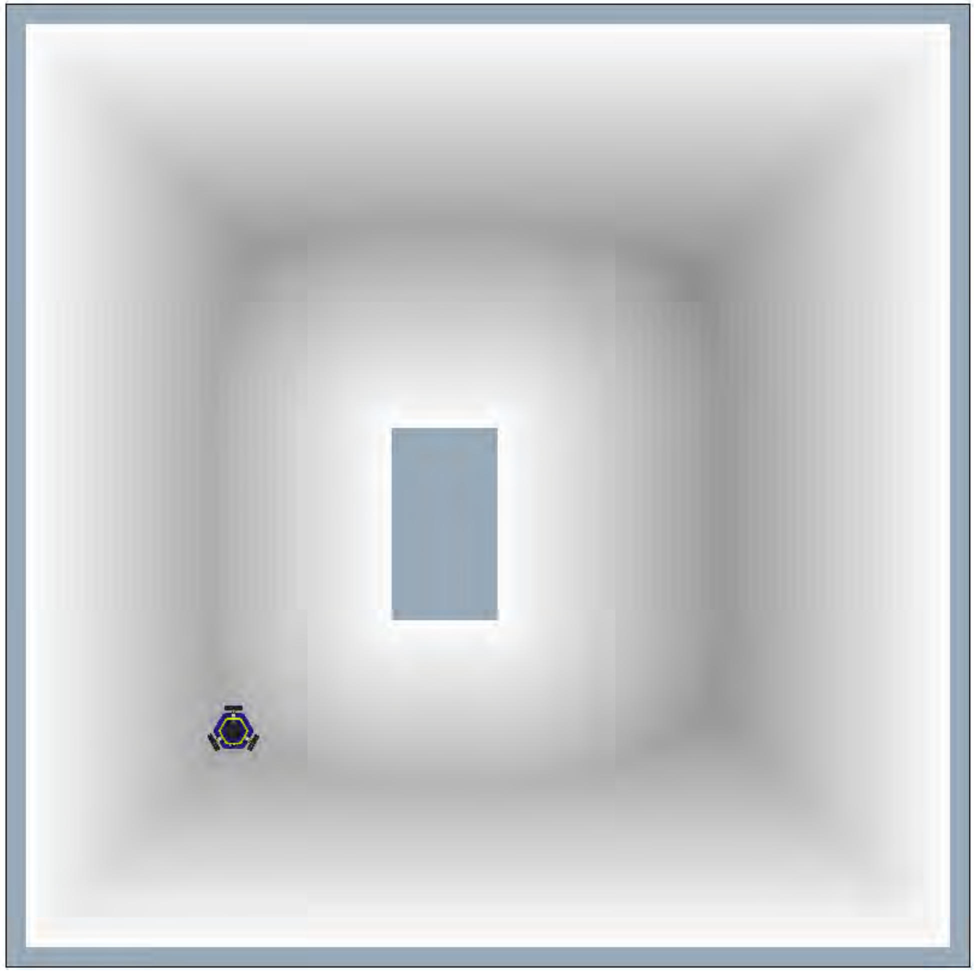


Computing the Distance Transform

You will implement this “slow” distance transform using the Euclidean distance in P2.2.

```
void distanceTransformSlow(GridGraph& graph)
{
    /**
     * TODO (P2): Perform a distance transform by finding the distance to the
     * nearest occupied cell for each unoccupied cell. Calculate the distance
     * to the nearest cell by looping through all the occupied cells in the
     * graph.
     *
     * Store the result in the vector graph.obstacle_distances.
     **/
}
```

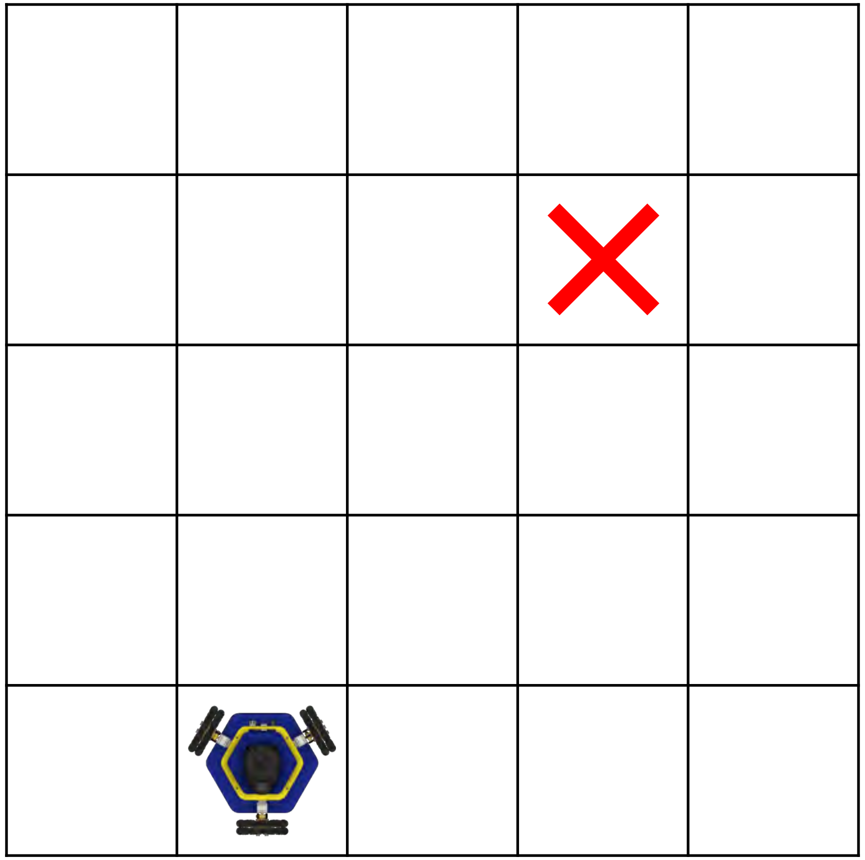
src/potential_field/distance_transform.cpp



The Manhattan Distance

Another way to compute the distance between cells is the Manhattan distance.

We can get a faster distance transform algorithm if we use the Manhattan Distance.



The Manhattan Distance

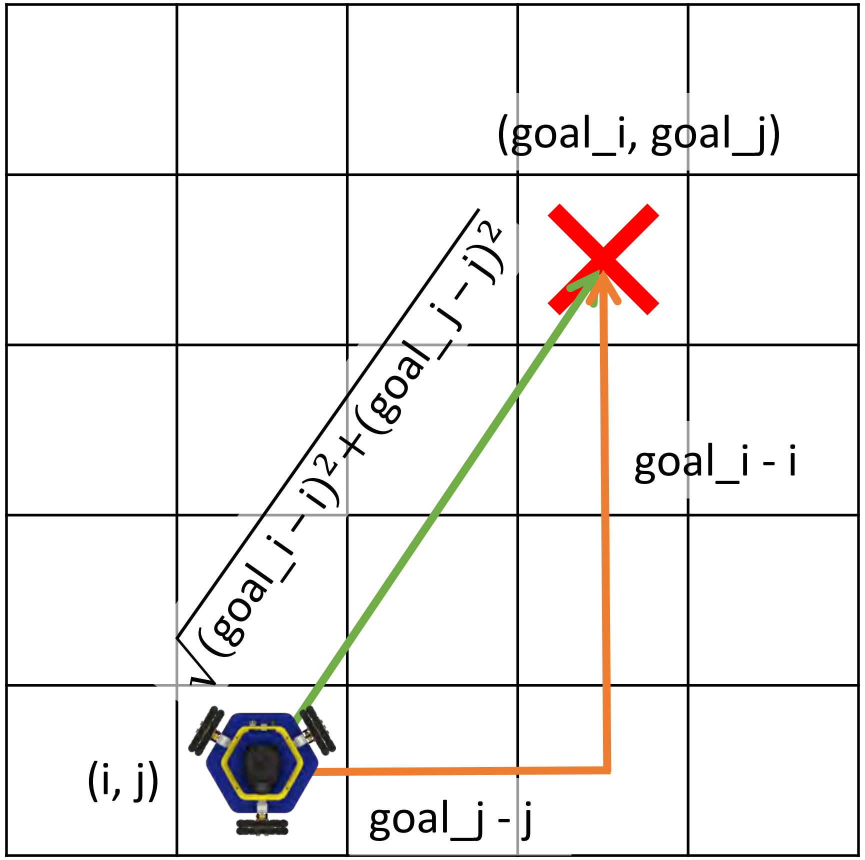
Another way to compute the distance between cells is the Manhattan distance.

Euclidean:

$$\text{dist} = \sqrt{(\text{goal\_i} - \text{i})^2 + (\text{goal\_j} - \text{j})^2}$$

Manhattan:

$$\text{dist} = |\text{goal\_i} - \text{i}| + |\text{goal\_j} - \text{j}|$$



The name "Manhattan distance" comes from the grid layout of city blocks in Manhattan. The shortest path from one location to another requires walking along the grid.

The Manhattan Distance

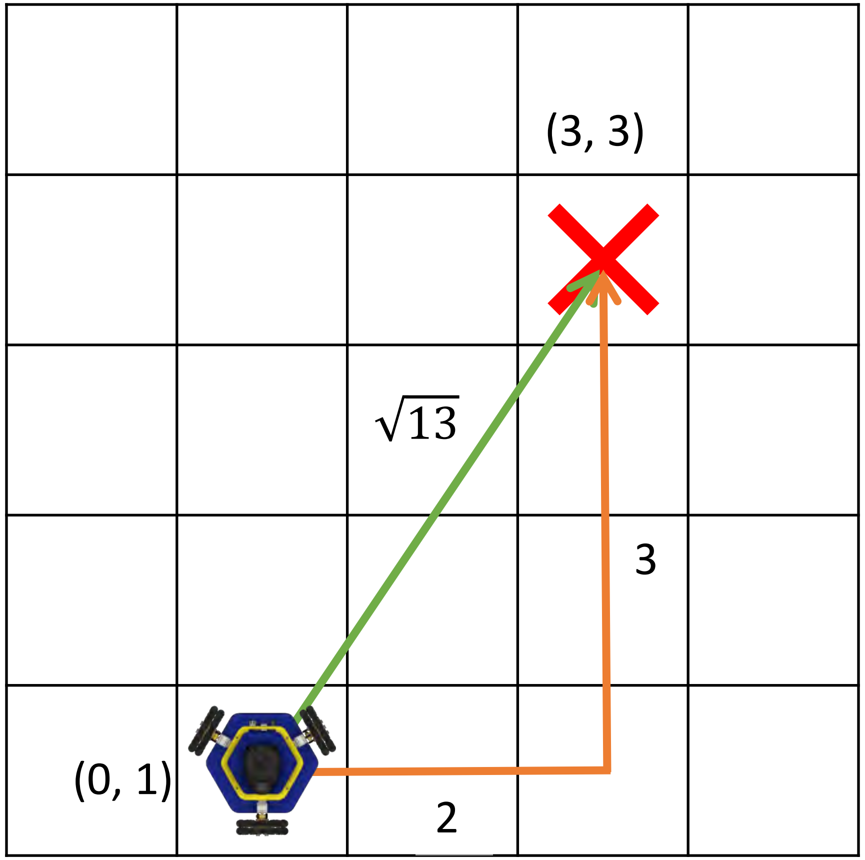
Another way to compute the distance between cells is the Manhattan distance.

Euclidean:

$$\text{dist} = \sqrt{(3-0)^2 + (3-1)^2}$$
$$= \sqrt{(3)^2 + (2)^2} = \sqrt{13}$$

Manhattan:

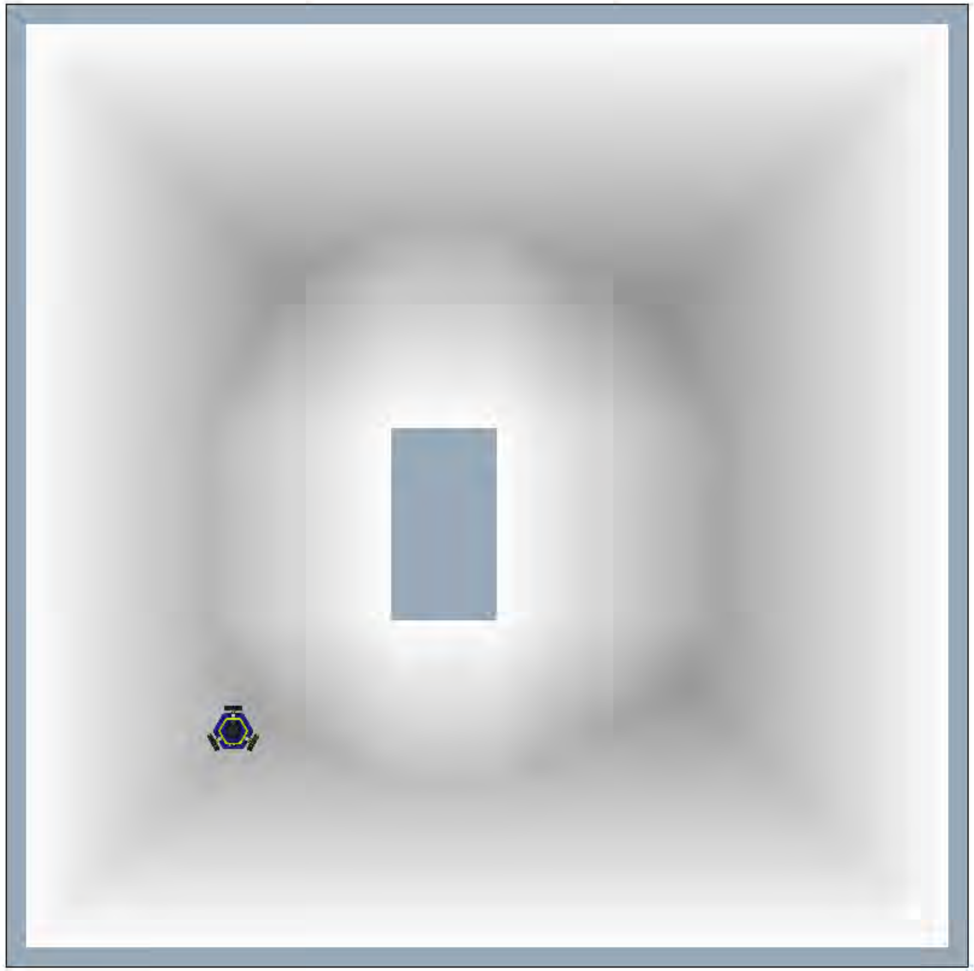
$$\text{dist} = |3-0| + |3-1|$$
$$= |3| + |2| = 5$$



Manhattan Distance Transform

The Manhattan distance transform involves computing the Manhattan distance from each cell to the nearest occupied cell.

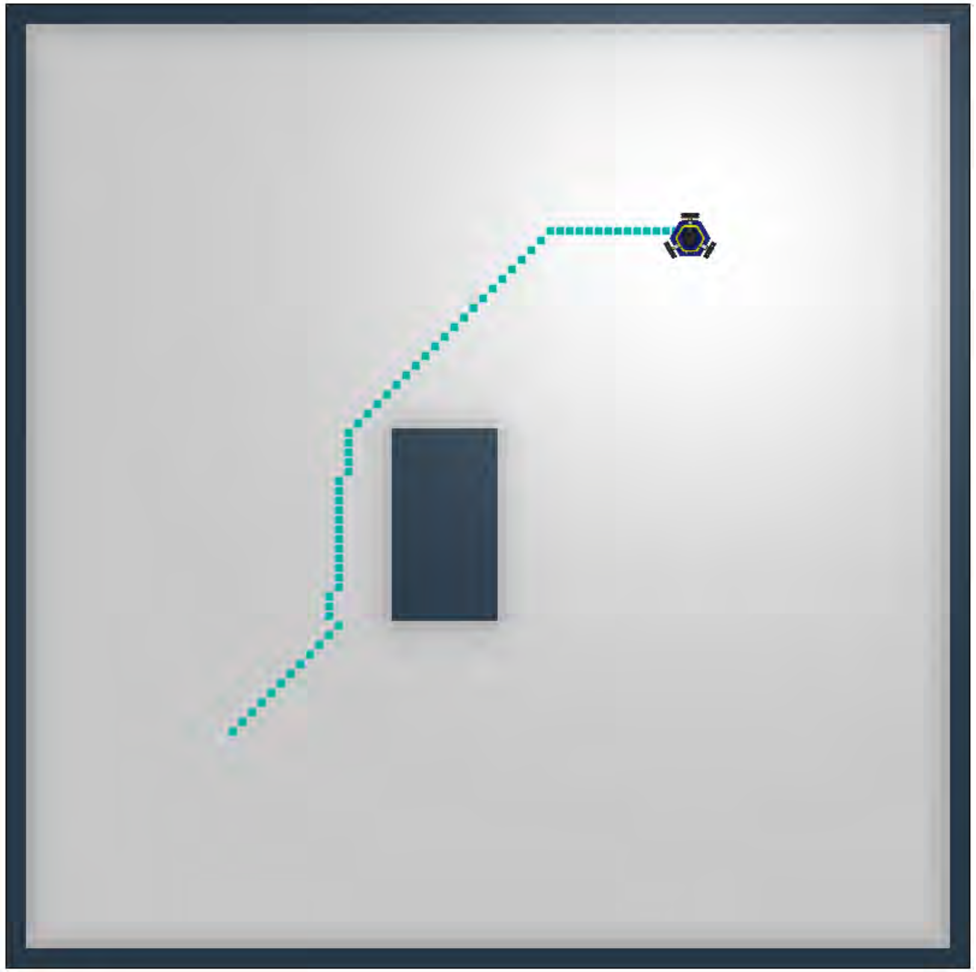
The Manhattan distance transform is less “smooth” than the Euclidean version, but they look similar.



Example: Manhattan Distance Transform

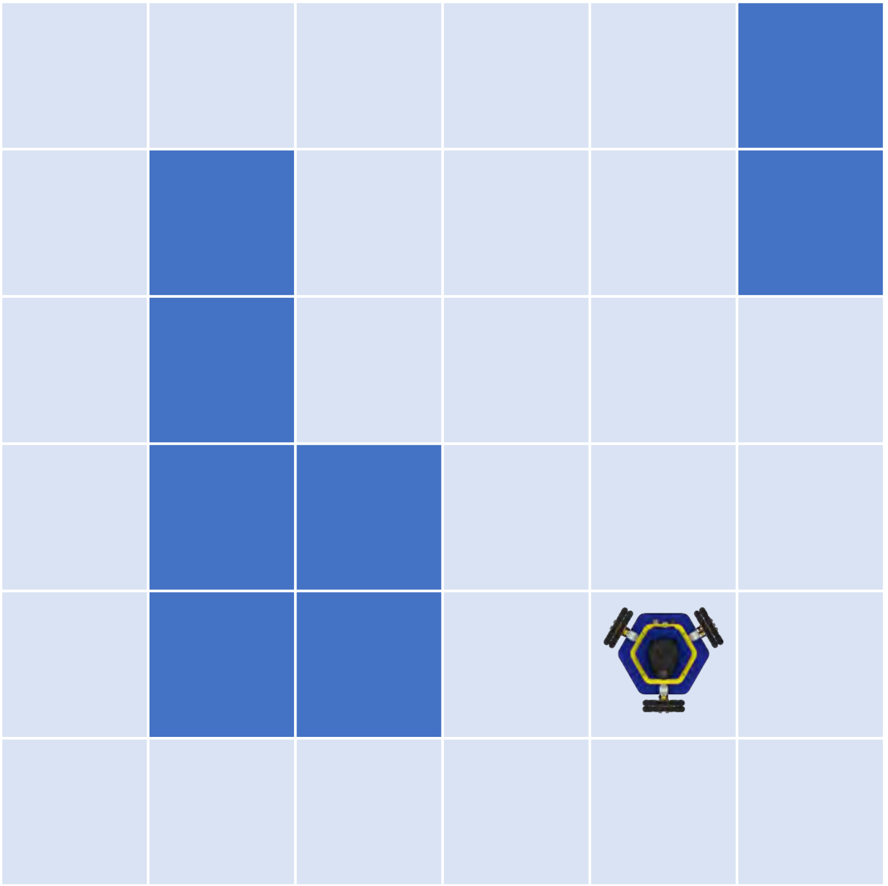
It turns out that the Manhattan distance transform is *good enough* to do potential field navigation.

And the algorithm for calculating it is much faster!



Manhattan Distance Transform Algorithm

Imagine our robot has a small 2D map that looks like this one.



Manhattan Distance Transform Algorithm

Imagine our robot has a small 2D map that looks like this one.

We can convert this to a binary map which has value 1 if the cell is occupied, and 0 is the cell is free.

0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	 0	0
0	0	0	0	0	0

Manhattan Distance Transform Algorithm

The Manhattan distance transform needs to scan along the rows and columns to find the nearest obstacle.

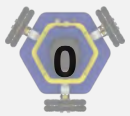
0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	0	3	0

Manhattan Distance Transform Algorithm

The Manhattan distance transform needs to scan along the rows and columns to find the nearest obstacle.

We will start with the simpler case of a 1D distance transform by looking at one row.

0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	0	0	0

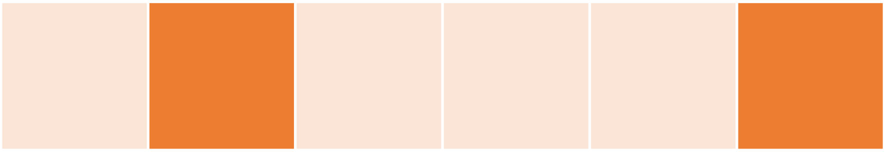


1D Manhattan Distance Transform

We can treat one row of the map as a 1D binary map:

0	1	0	0	0	1

The distance transform is the distance from each free cell (0) to the nearest occupied cell (1). We can write down the answer by inspection:



1D Manhattan Distance Transform

We can treat one row of the map as a 1D binary map:

0	1	0	0	0	1

The distance transform is the distance from each free cell (0) to the nearest occupied cell (1). We can write down the answer by inspection:

	0				0

These cells are occupied, so their distance to the nearest occupied cell is zero

1D Manhattan Distance Transform

We can treat one row of the map as a 1D binary map:

0	1	0	0	0	1

The distance transform is the distance from each free cell (0) to the nearest occupied cell (1). We can write down the answer by inspection:

1	0				0

One cell away from an occupied cell

1D Manhattan Distance Transform

We can treat one row of the map as a 1D binary map:

0	1	0	0	0	1

The distance transform is the distance from each free cell (0) to the nearest occupied cell (1). We can write down the answer by inspection:

1	0	1			0

One cell away from
an occupied cell

1D Manhattan Distance Transform

We can treat one row of the map as a 1D binary map:

0	1	0	0	0	1

The distance transform is the distance from each free cell (0) to the nearest occupied cell (1). We can write down the answer by inspection:

1	0	1		1	0

One cell away from an occupied cell

1D Manhattan Distance Transform

We can treat one row of the map as a 1D binary map:

0	1	0	0	0	1

The distance transform is the distance from each free cell (0) to the nearest occupied cell (1). We can write down the answer by inspection:

1	0	1	2	1	0

Two cells away from
an occupied cell

We need an algorithm to compute the distance transform on a computer.

1D Manhattan Distance Transform

1. Initialize.
    - For each cell, set distance transform DT to 0 if the cell is occupied, and infinity if the cell is free

0	1	0	0	0	1

Initialization step:

DT =

∞	0	∞	∞	∞	0

1D Manhattan Distance Transform

1. Initialize to zero or infinity.
2. Forward pass:
   - For cells i=1 to N-1:

     DT[i] = min(DT[i], DT[i-1] + 1)

0	1	0	0	0	1

Initialization step:

DT =

∞	0	∞	∞	∞	0

Forward pass:

DT =

∞	0	∞	∞	∞	0

DT[1] = min(0, ∞+1)

1D Manhattan Distance Transform

1. Initialize to zero or infinity.
2. Forward pass:
   - For cells i=1 to N-1:

     DT[i] = min(DT[i], DT[i-1] + 1)

0	1	0	0	0	1

Initialization step:

DT =

∞	0	∞	∞	∞	0

Forward pass:

DT =

∞	0	$\boxed{1}$	∞	∞	0

DT[2] = min(∞, 0+1)

1D Manhattan Distance Transform

1. Initialize to zero or infinity.
2. Forward pass:
   - For cells i=1 to N-1:

     DT[i] = min(DT[i], DT[i-1] + 1)

0	**1**	**0**	**0**	**0**	**1**

Initialization step:

DT =

∞	**0**	∞	∞	∞	**0**

Forward pass:

DT =

∞	**0**	**1**	$\boxed{\mathbf{2}}$	∞	**0**

DT[3] = min(∞, 1+1)

1D Manhattan Distance Transform

1. Initialize to zero or infinity.
2. Forward pass:
   - For cells i=1 to N-1:

     DT[i] = min(DT[i], DT[i-1] + 1)

0	**1**	**0**	**0**	**0**	**1**

Initialization step:

DT =

∞	**0**	∞	∞	∞	**0**

Forward pass:

DT =

∞	**0**	**1**	**2**	$\boxed{\mathbf{3}}$	**0**

DT[4] = min(∞, 2+1)

1D Manhattan Distance Transform

1. Initialize to zero or infinity.
2. Forward pass:
   - For cells i=1 to N-1:

     DT[i] = min(DT[i], DT[i-1] + 1)

0	1	0	0	0	1

Initialization step:

DT =

∞	0	∞	∞	∞	0

Forward pass:

DT =

∞	0	1	2	3	$\boxed{0}$

DT[5] = min(0, 3+1)

1D Manhattan Distance Transform

1. Initialize to zero or infinity.
2. Forward pass:
   - For cells i=1 to N-1:

     DT[i] = min(DT[i], DT[i-1] + 1)
3. Backward pass:
   - For cells i=N-2 to 0:

     DT[i] = min(DT[i], DT[i+1] + 1)

0	1	0	0	0	1

Initialization step:

DT =	∞	0	∞	∞	∞	0

Forward pass:

DT =	∞	0	1	2	3	0

Backward pass:

DT =	∞	0	1	2	$\boxed{1}$	0

DT[4] = min(3, 0+1)

1D Manhattan Distance Transform

1. Initialize to zero or infinity.
2. Forward pass:
   - For cells i=1 to N-1:

     DT[i] = min(DT[i], DT[i-1] + 1)
3. Backward pass:
   - For cells i=N-2 to 0:

     DT[i] = min(DT[i], DT[i+1] + 1)

0	1	0	0	0	1

Initialization step:

DT =

∞	0	∞	∞	∞	0

Forward pass:

DT =

∞	0	1	2	3	0

Backward pass:

DT =

∞	0	1	$\boxed{2}$	1	0

DT[3] = min(2, 1+1)

1D Manhattan Distance Transform

1. Initialize to zero or infinity.
2. Forward pass:
   - For cells i=1 to N-1:

     DT[i] = min(DT[i], DT[i-1] + 1)
3. Backward pass:
   - For cells i=N-2 to 0:

     DT[i] = min(DT[i], DT[i+1] + 1)

0	1	0	0	0	1

Initialization step:

DT =

∞	0	∞	∞	∞	0

Forward pass:

DT =

∞	0	1	2	3	0

Backward pass:

DT =

∞	0	$\boxed{1}$	2	1	0

DT[2] = min(1, 2+1)

1D Manhattan Distance Transform

1. Initialize to zero or infinity.
2. Forward pass:
   - For cells i=1 to N-1:

     DT[i] = min(DT[i], DT[i-1] + 1)
3. Backward pass:
   - For cells i=N-2 to 0:

     DT[i] = min(DT[i], DT[i+1] + 1)

0	1	0	0	0	1

Initialization step:

DT =	∞	0	∞	∞	∞	0

Forward pass:

DT =	∞	0	1	2	3	0

Backward pass:

DT =	∞	0	1	2	1	0

DT[1] = min(0, 1+1)

1D Manhattan Distance Transform

1. Initialize to zero or infinity.
2. Forward pass:
   - For cells i=1 to N-1:

     DT[i] = min(DT[i], DT[i-1] + 1)
3. Backward pass:
   - For cells i=N-2 to 0:

     DT[i] = min(DT[i], DT[i+1] + 1)

0	1	0	0	0	1

Initialization step:

DT =	∞	0	∞	∞	∞	0

Forward pass:

DT =	∞	0	1	2	3	0

Backward pass:

DT =	$\boxed{1}$	0	1	2	1	0

DT[0] = min(∞, 0+1)

1D Manhattan Distance Transform

1. Initialize to zero or infinity. } N loops
2. Forward pass:
   - For cells i=1 to N-1:

     DT[i] = min(DT[i], DT[i-1] + 1)

   N-1 loops
3. Backward pass:
   - For cells i=N-2 to 0:

     DT[i] = min(DT[i], DT[i+1] + 1)

   N-1 loops

How many computations did we do?

Total: N + 2 * (N – 1) ≈ 3N

0	1	0	0	0	1

Initialization step:

DT =	∞	0	∞	∞	∞	0

Forward pass:

DT =	∞	0	1	2	3	0

Backward pass:

DT =	1	0	1	2	1	0

1D Manhattan Distance Transform

This algorithm is faster, especially for large graphs!

Note: Manhattan distance and Euclidean distance are the same in 1D.

Oct 13 In-Class Activity: 1D & 2D Manhattan distance transform.

0	1	0	0	0	1

Initialization step:

DT =

∞	0	∞	∞	∞	0

Forward pass:

DT =

∞	0	1	2	3	0

Backward pass:

DT =

1	0	1	2	1	0

2D Manhattan Distance Transform

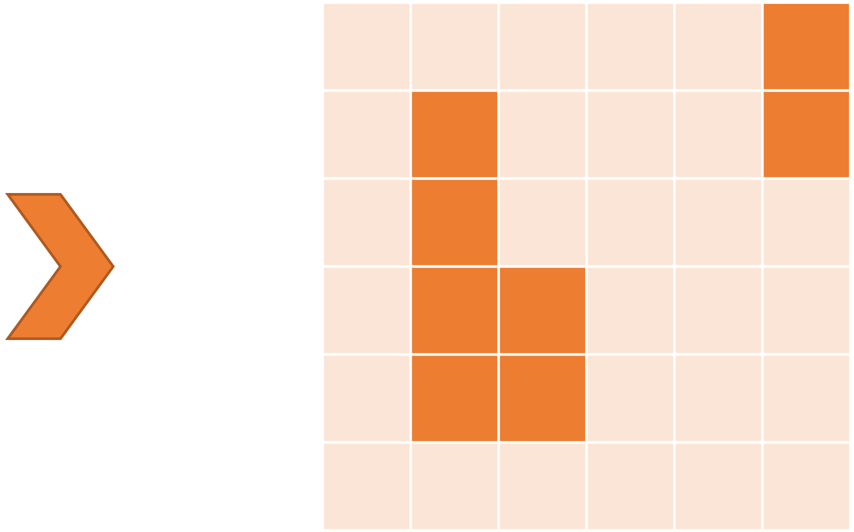
Back to our 2D map…

0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	0	0	0

2D Manhattan Distance Transform

Let’s do our 2D Manhattan distance transform by inspection:

0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	0	0	0



2D Manhattan Distance Transform

Let’s do our 2D Manhattan distance transform by inspection:

0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	0	0	0

					0
	0				0
	0				
	0	0			
	0	0			

All the occupied cells have value zero.

2D Manhattan Distance Transform

Let’s do our 2D Manhattan distance transform by inspection:

0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	0	0	0

					0
	0				0
	0				
	0	0			
	0	0			
$\boxed{2}$					

1

1

DT[0, 0] = 1 + 1

2D Manhattan Distance Transform

Let’s do our 2D Manhattan distance transform by inspection:

0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	0	0	0

					0
	0				0
	0				
	0	0			
1	0	0			
2	$\boxed{1}$				

DT[0, 1] = 1 + 0

2D Manhattan Distance Transform

Let’s do our 2D Manhattan distance transform by inspection:

0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	0	0	0

					0
	0				0
	0				
	0	0			
	0	0			
2	1	$\boxed{1}$			

1

DT[0, 2] = 1 + 0

2D Manhattan Distance Transform

Let’s do our 2D Manhattan distance transform by inspection:

0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	0	0	0

					0
	0				0
	0				
	0	0			
	0	0			
2	1	1	$\boxed{2}$		

1

1

DT[0, 3] = 1 + 1

2D Manhattan Distance Transform

Let’s do our 2D Manhattan distance transform by inspection:

0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	0	0	0

					0
	0				0
	0				
	0	0			
	0	0			
2	1	1	2	3	

1

2

DT[0, 4] = 2 + 1

2D Manhattan Distance Transform

Let’s do our 2D Manhattan distance transform by inspection:

0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	0	0	0

					0
	0				0
	0				
	0	0			
	0	0			
2	1	1	2	3	$\boxed{4}$

1

3

DT[0, 5] = 3 + 1

2D Manhattan Distance Transform

Let’s do our 2D Manhattan distance transform by inspection:

0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	0	0	0

2	1	2	2	1	0
1	0	1	2	1	0
1	0	1	2	2	1
1	0	0	1	2	2
1	0	0	1	2	3
2	1	1	2	3	4

It turns out that we can use a modification of the algorithm for the 1D transform to compute our 2D Manhattan distance transform.

2D Manhattan Distance Transform

1. Initialize: Set occupied cells to zero and free cells to infinity.

0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	0	0	0

∞	∞	∞	∞	∞	0
∞	0	∞	∞	∞	0
∞	0	∞	∞	∞	∞
∞	0	0	∞	∞	∞
∞	0	0	∞	∞	∞
∞	∞	∞	∞	∞	∞

2D Manhattan Distance Transform

1. Initialize: Set occupied cells to zero and free cells to infinity.
2. Forward pass:
   - For i=1 to N-1:

     DT[i] = min(DT[i], Bottom neighbor + 1, Left neighbor +1)

     If there is not bottom or left neighbor, ignore.

0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	0	0	0

∞	∞	∞	∞	∞	0
∞	0	∞	∞	∞	0
∞	0	∞	∞	∞	∞
∞	0	0	∞	∞	∞
∞	0	0	∞	∞	∞
∞	∞	∞	∞	∞	∞

DT[0, 1] = min(∞, ∞+1)

2D Manhattan Distance Transform

1. Initialize: Set occupied cells to zero and free cells to infinity.
2. Forward pass:
   - For i=1 to N-1:

     DT[i] = min(DT[i], Bottom neighbor + 1, Left neighbor +1)

     If there is not bottom or left neighbor, ignore.

0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	0	0	0

∞	∞	∞	∞	∞	0
∞	0	∞	∞	∞	0
∞	0	∞	∞	∞	∞
∞	0	0	∞	∞	∞
∞	0	0	∞	∞	∞
∞	∞	∞	∞	∞	∞

DT[0, 2] = min(∞, ∞+1)

2D Manhattan Distance Transform

1. Initialize: Set occupied cells to zero and free cells to infinity.
2. Forward pass:
   - For i=1 to N-1:

     DT[i] = min(DT[i], Bottom neighbor + 1, Left neighbor +1)

     If there is not bottom or left neighbor, ignore.

0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	0	0	0

∞	∞	∞	∞	∞	0
∞	0	∞	∞	∞	0
∞	0	∞	∞	∞	∞
∞	0	0	∞	∞	∞
∞	0	0	∞	∞	∞
∞	∞	∞	∞	∞	∞

DT[1, 0] = min(∞, ∞+1)

2D Manhattan Distance Transform

1. Initialize: Set occupied cells to zero and free cells to infinity.
2. Forward pass:
   - For i=1 to N-1:

     DT[i] = min(DT[i], Bottom neighbor + 1, Left neighbor +1)

     If there is not bottom or left neighbor, ignore.

0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	0	0	0

∞	∞	∞	∞	∞	0
∞	0	∞	∞	∞	0
∞	0	∞	∞	∞	∞
∞	0	0	∞	∞	∞
∞	0	0	∞	∞	∞
∞	∞	∞	∞	∞	∞

DT[1, 1] = min(0, ∞+1, ∞+1)

2D Manhattan Distance Transform

1. Initialize: Set occupied cells to zero and free cells to infinity.
2. Forward pass:
   - For i=1 to N-1:

     DT[i] = min(DT[i], Bottom neighbor + 1, Left neighbor +1)

     If there is not bottom or left neighbor, ignore.

0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	0	0	0

∞	∞	∞	∞	∞	0
∞	0	∞	∞	∞	0
∞	0	∞	∞	∞	∞
∞	0	0	∞	∞	∞
∞	0	0	∞	∞	∞
∞	∞	∞	∞	∞	∞

DT[1, 2] = min(0, 0+1, ∞+1)

2D Manhattan Distance Transform

1. Initialize: Set occupied cells to zero and free cells to infinity.
2. Forward pass:
   - For i=1 to N-1:

     DT[i] = min(DT[i], Bottom neighbor + 1, Left neighbor +1)

     If there is not bottom or left neighbor, ignore.

0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	0	0	0

∞	∞	∞	∞	∞	0
∞	0	∞	∞	∞	0
∞	0	∞	∞	∞	∞
∞	0	0	∞	∞	∞
∞	0	0	1	∞	∞
∞	∞	∞	∞	∞	∞

DT[1, 3] = min(∞, 0+1, ∞+1)

2D Manhattan Distance Transform

1. Initialize: Set occupied cells to zero and free cells to infinity.
2. Forward pass:
   - For i=1 to N-1:

     DT[i] = min(DT[i], Bottom neighbor + 1, Left neighbor +1)

     If there is not bottom or left neighbor, ignore.

0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	0	0	0

∞	∞	∞	∞	∞	0
∞	0	∞	∞	∞	0
∞	0	∞	∞	∞	∞
∞	0	0	∞	∞	∞
∞	0	0	1	2	∞
∞	∞	∞	∞	∞	∞

DT[1, 4] = min(∞, 1+1, ∞+1)

2D Manhattan Distance Transform

1. Initialize: Set occupied cells to zero and free cells to infinity.
2. Forward pass:
   - For i=1 to N-1:

     DT[i] = min(DT[i], Bottom neighbor + 1, Left neighbor +1)

     If there is not bottom or left neighbor, ignore.

0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	0	0	0

∞	∞	∞	∞	∞	0
∞	0	∞	∞	∞	0
∞	0	∞	∞	∞	∞
∞	0	0	∞	∞	∞
∞	0	0	1	2	3
∞	∞	∞	∞	∞	∞

DT[1, 4] = min(∞, 2+1, ∞+1)

2D Manhattan Distance Transform

1. Initialize: Set occupied cells to zero and free cells to infinity.
2. Forward pass:
   - For i=1 to N-1:

     DT[i] = min(DT[i], Bottom neighbor + 1, Left neighbor +1)

     If there is not bottom or left neighbor, ignore.

0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	0	0	0

∞	1	2	3	4	0
∞	0	1	2	3	0
∞	0	1	2	3	4
∞	0	0	1	2	3
∞	0	0	1	2	3
∞	∞	∞	∞	∞	∞

2D Manhattan Distance Transform

1. Initialize: Set occupied cells to zero and free cells to infinity.
2. Forward pass:
   - For i=1 to N-1:

     DT[i] = min(DT[i], Bottom neighbor + 1, Left neighbor +1)

     If there is not bottom or left neighbor, ignore.
3. Backward pass:
   - For i=N-2 to 0:

     DT[i] = min(DT[i], Top neighbor + 1, Right neighbor +1)

     If there is no top or right neighbor, ignore.

0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	0	0	0

∞	1	2	3	1	0
∞	0	1	2	3	0
∞	0	1	2	3	4
∞	0	0	1	2	3
∞	0	0	1	2	3
∞	∞	∞	∞	∞	∞

DT[5, 4] = min(4, 0+1)

2D Manhattan Distance Transform

1. Initialize: Set occupied cells to zero and free cells to infinity.
2. Forward pass:
   - For i=1 to N-1:

     DT[i] = min(DT[i], Bottom neighbor + 1, Left neighbor +1)

     If there is not bottom or left neighbor, ignore.
3. Backward pass:
   - For i=N-2 to 0:

     DT[i] = min(DT[i], Top neighbor + 1, Right neighbor +1)

     If there is no top or right neighbor, ignore.

0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	0	0	0

∞	1	2	2	1	0
∞	0	1	2	3	0
∞	0	1	2	3	4
∞	0	0	1	2	3
∞	0	0	1	2	3
∞	∞	∞	∞	∞	∞

DT[5, 3] = min(3, 1+1)

2D Manhattan Distance Transform

1. Initialize: Set occupied cells to zero and free cells to infinity.
2. Forward pass:
   - For i=1 to N-1:

     DT[i] = min(DT[i], Bottom neighbor + 1, Left neighbor +1)

     If there is not bottom or left neighbor, ignore.
3. Backward pass:
   - For i=N-2 to 0:

     DT[i] = min(DT[i], Top neighbor + 1, Right neighbor +1)

     If there is no top or right neighbor, ignore.

0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	0	0	0

∞	1	$\boxed{2}$	$\boxed{2}$	1	0
∞	0	1	2	3	0
∞	0	1	2	3	4
∞	0	0	1	2	3
∞	0	0	1	2	3
∞	∞	∞	∞	∞	∞

DT[5, 2] = min(2, 2+1)

2D Manhattan Distance Transform

1. Initialize: Set occupied cells to zero and free cells to infinity.
2. Forward pass:
   - For i=1 to N-1:

     DT[i] = min(DT[i], Bottom neighbor + 1, Left neighbor +1)

     If there is not bottom or left neighbor, ignore.
3. Backward pass:
   - For i=N-2 to 0:

     DT[i] = min(DT[i], Top neighbor + 1, Right neighbor +1)

     If there is no top or right neighbor, ignore.

0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	0	0	0

∞	1	2	2	1	0
∞	0	1	2	3	0
∞	0	1	2	3	4
∞	0	0	1	2	3
∞	0	0	1	2	3
∞	∞	∞	∞	∞	∞

DT[5, 1] = min(1, 2+1)

2D Manhattan Distance Transform

1. Initialize: Set occupied cells to zero and free cells to infinity.
2. Forward pass:
   - For i=1 to N-1:

     DT[i] = min(DT[i], Bottom neighbor + 1, Left neighbor +1)

     If there is not bottom or left neighbor, ignore.
3. Backward pass:
   - For i=N-2 to 0:

     DT[i] = min(DT[i], Top neighbor + 1, Right neighbor +1)

     If there is no top or right neighbor, ignore.

0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	0	0	0

2	1	2	2	1	0
∞	0	1	2	3	0
∞	0	1	2	3	4
∞	0	0	1	2	3
∞	0	0	1	2	3
∞	∞	∞	∞	∞	∞

DT[5, 0] = min(∞, 1+1)

2D Manhattan Distance Transform

1. Initialize: Set occupied cells to zero and free cells to infinity.
2. Forward pass:
   - For i=1 to N-1:

     DT[i] = min(DT[i], Bottom neighbor + 1, Left neighbor +1)

     If there is not bottom or left neighbor, ignore.
3. Backward pass:
   - For i=N-2 to 0:

     DT[i] = min(DT[i], Top neighbor + 1, Right neighbor +1)

     If there is no top or right neighbor, ignore.

0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	0	0	0

2	1	2	2	1	0
∞	0	1	2	3	0
∞	0	1	2	3	4
∞	0	0	1	2	3
∞	0	0	1	2	3
∞	∞	∞	∞	∞	∞

DT[4, 5] = min(0, 0+1)

2D Manhattan Distance Transform

1. Initialize: Set occupied cells to zero and free cells to infinity.
2. Forward pass:
    - For i=1 to N-1:

      DT[i] = min(DT[i], Bottom neighbor + 1, Left neighbor +1)

      If there is not bottom or left neighbor, ignore.
3. Backward pass:
    - For i=N-2 to 0:

      DT[i] = min(DT[i], Top neighbor + 1, Right neighbor +1)

      If there is no top or right neighbor, ignore.

0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	0	0	0

2	1	2	2	1	0
∞	0	1	2	1	0
∞	0	1	2	3	4
∞	0	0	1	2	3
∞	0	0	1	2	3
∞	∞	∞	∞	∞	∞

DT[4, 5] = min(3, 1+1, 0+1)

2D Manhattan Distance Transform

1. Initialize: Set occupied cells to zero and free cells to infinity.
2. Forward pass:
   - For i=1 to N-1:

     DT[i] = min(DT[i], Bottom neighbor + 1, Left neighbor +1)

     If there is not bottom or left neighbor, ignore.
3. Backward pass:
   - For i=N-2 to 0:

     DT[i] = min(DT[i], Top neighbor + 1, Right neighbor +1)

     If there is no top or right neighbor, ignore.

0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	0	0	0

2	1	2	2	1	0
∞	0	1	2	1	0
∞	0	1	2	3	4
∞	0	0	1	2	3
∞	0	0	1	2	3
∞	∞	∞	∞	∞	∞

DT[4, 5] = min(2, 2+1, 1+1)

2D Manhattan Distance Transform

1. Initialize: Set occupied cells to zero and free cells to infinity.
2. Forward pass:
   - For i=1 to N-1:

     DT[i] = min(DT[i], Bottom neighbor + 1, Left neighbor +1)

     If there is not bottom or left neighbor, ignore.
3. Backward pass:
   - For i=N-2 to 0:

     DT[i] = min(DT[i], Top neighbor + 1, Right neighbor +1)

     If there is no top or right neighbor, ignore.

0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	0	0	0

2	1	2	2	1	0
1	0	1	2	1	0
1	0	1	2	2	1
1	0	0	1	2	2
1	0	0	1	2	3
2	1	1	2	3	4

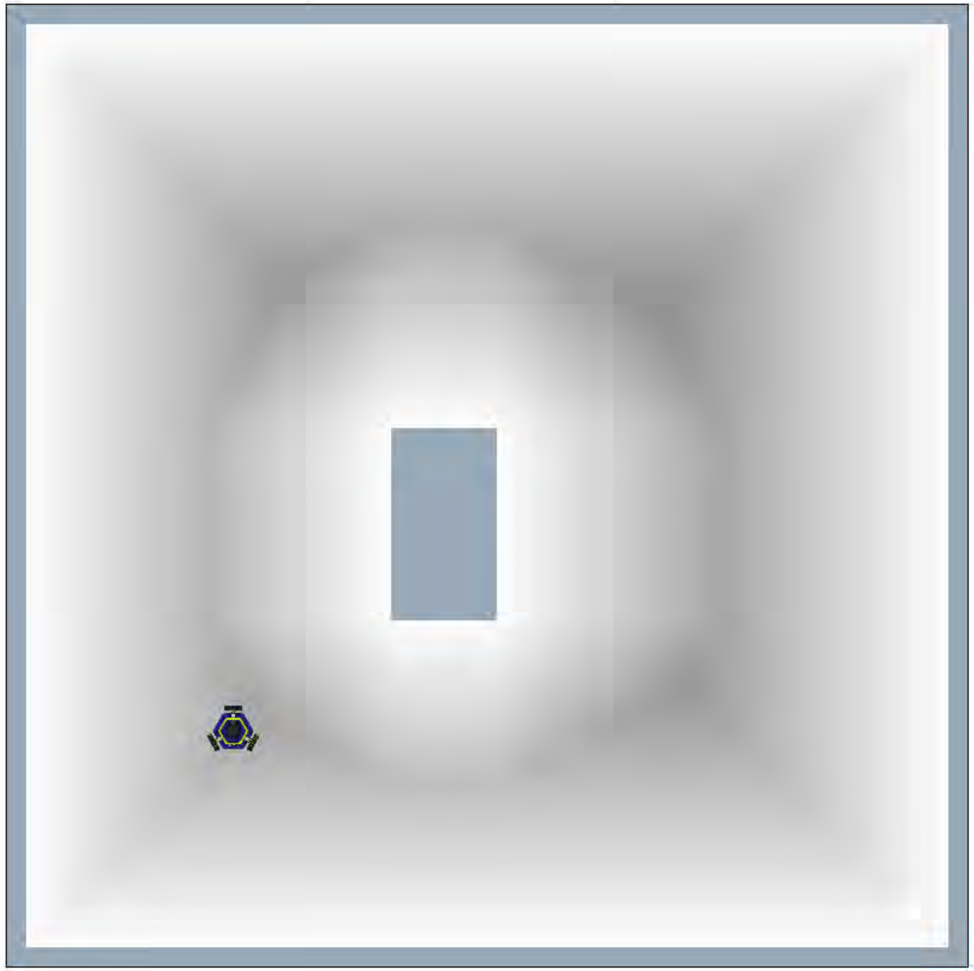
How many computations did we do? Total: N + 2 * (N – 1) ≈ 3N

Manhattan Distance Transform

You will compute the Manhattan distance transform in P2.2.

```
void distanceTransformManhattan(GridGraph& graph)
{
    /**
     * TODO (P2): Perform a distance transform using the Manhattan distance
     * transform algorithm over a 2D grid.
     *
     * Store the result in the vector graph.obstacle_distances.
     **/
}
```

src/potential_field/distance_transform.cpp



Project 2: Potential Field Navigation

- [x] Build a map of environment
- [x] Form attraction potential to goal
- [x] Form repulsion potentials away from obstacles
- [ ] Add potentials together into potential field   Next time!
- [x] Local search over potential field to navigate