Numerical Potential Field Path Planning Tutorial

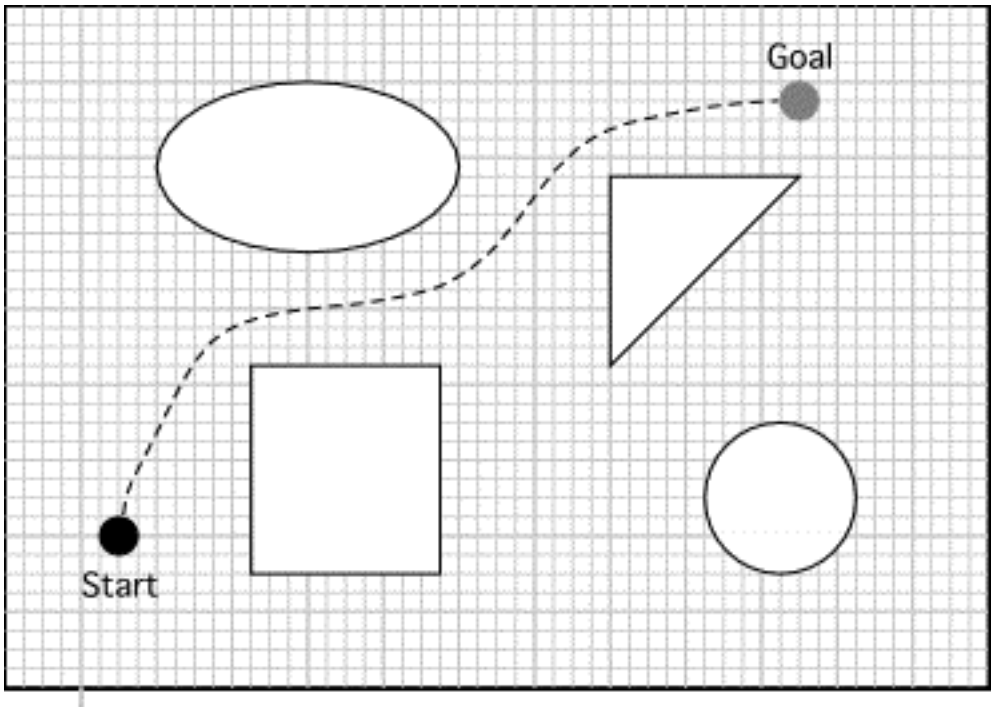
Matt Greytak
December 9, 2005

Contents:

Part 1: Introduction

A fast, effective path planning algorithm is essential for autonomous mobile robots. Terrestrial vehicles, spacecraft, and even robotic manipulators must be able to navigate through a known or unknown environment without colliding with obstacles, getting stuck, or sustaining damage.

The figure below shows a reasonable path through a set of obstacles.



A typical path planning problem with a reasonable solution

There are several path planning algorithms that can be applied to our problem. One is called a Visibility Graph method: first, all of the objects are expanded by a distance equal to the largest radius of the robot.

This creates a buffer around all of the obstacles. Next a set of straight lines are drawn between all the corners of the expanded obstacles and the start and goal positions. A search is performed to find the shortest distance from the start to the goal; informed search methods such as A* or Branch and Bound may be used. The resulting path is piecewise linear between the corners of the expanded obstacles. Unfortunately this algorithm creates paths that pass very close to obstacles (by design). A hallway-navigating robot using this algorithm would hug the walls and always aim for the corners.

Another algorithm, the Voronoi Graph method, finds paths that are equidistant from at least two points on the nearest obstacles. These paths are not easy to develop, but they stay as far away from obstacles as possible. As before a search is performed to find the one path that most directly links the start to the goal. A hallway-navigating robot using this algorithm would always stay in the middle of the hallway and never cut corners.

Ideally we would like our path to stay reasonably far from obstacles, but not necessarily exactly in between their boundaries. We would like it to approach corners when necessary but not otherwise, and never get too close to them. All of these goals are satisfied by a third path planning algorithm: the Numerical Potential Field method.

In the third algorithm, an artificial potential field is set up in the space; that is, each point in the space is assigned a scalar value. The value at the goal point is set to be 0 and the value of the potential at all other points is positive. The potential at each point has two contributions: a goal force that causes the potential to increase with path distance from the goal, and an obstacle force that increases in inverse proportion to the distance to the nearest obstacle boundary. In other words, the potential is lowest at the goal, large at points far from the goal, and large at points next to obstacles. If the potential is suitably defined, then if a robot starts at any point in the space and always moves in the direction of the steepest negative potential slope, then the robot will move towards the goal while avoiding obstacles. The Potential Field algorithm is described in greater detail in the next two sections; in practice it produces paths that are a compromise between the Visibility Graph and Voronoi Graph methods.

While it takes some computational effort to compute the potential for the space, once the potential is known paths can be created on-the-fly from any point to the goal. If a robot is operating in a known environment then the potential needs to be computed only once for each goal and the robot will find the optimal path to the goal from any initial location without any extra computation. This has important implications for robustness: if the robot is pushed off the path due to disturbances or failures then the robot can continue to follow the steepest negative potential slope to reach the goal.

The numerical potential field path planner is guaranteed to produce a path even if the start or goal is placed in an obstacle. If there is no possible way to get from the start to the goal without passing through an obstacle then the path planner will generate a path through the obstacle, although if there is any alternative then the path will do that instead. For this reason it is important to make sure that there is some possible path, although there are ways around this restriction such as returning an error if the potential at the start point is too high.

The next section describes formally the problem inputs and outputs.

Numerical Potential Field Path Planning Tutorial

Matt Greytak
December 9, 2005

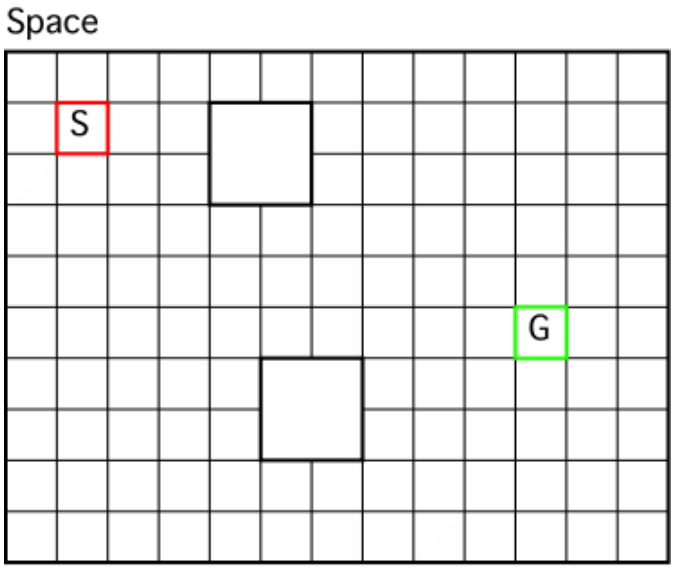
Contents:

Part 2: Problem Description

To formulate the path planning problem let us make a few assumptions. First we assume that the locations of the start position and goal position of the robot are known. Next we assume that the boundaries of the objects are known as well. We are not restricted to two or even three dimensions. Finally, assume that we can discretize the space and that our robot can follow a smoothed version of the resulting discrete trajectory.

Inputs and Outputs

As with any path planning problem, the input to the potential field path planner is a representation of the physical space in which the robot is to operate (a 2D Cartesian plane, for example), start and goal locations within that space, and the boundaries of all obstacles within the space that the robot must avoid. If the robot is operating in a closed environment then the enclosing boundaries can be treated as obstacles as well. The output of the planner is a path that connects the start location with the goal location. The output path is “valid” if it does not intersect with any obstacles.

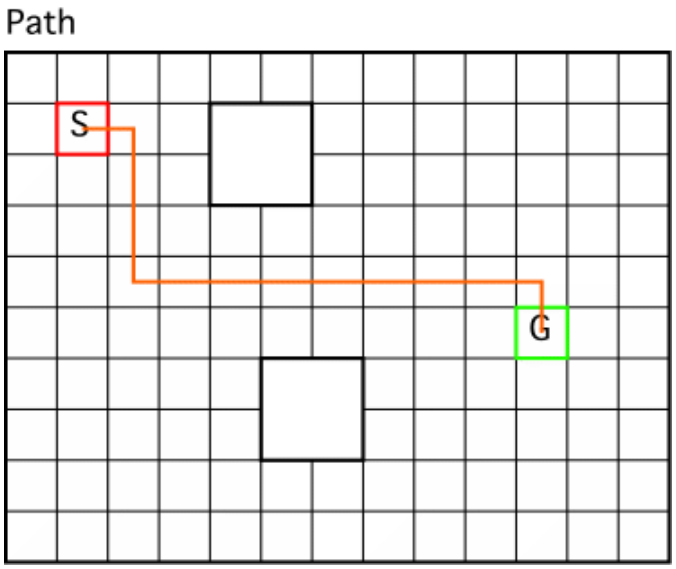


The input to the path planner: a space with obstacles, a start position, and a stop position.

The path returned by the planner is a set of points such as the following that connect the start to the goal.

Path Returned:
[2, 2]
[3, 2]
[3, 3]
[3, 4]
[3, 5]
[4, 5]
[5, 5]
[6, 5]
[7, 5]
[8, 5]
[9, 5]
[10, 5]
[11, 5]
[11, 6]

This particular path is drawn below.



Discretizing the Problem

For these potentials to be useful we must discretize the space. While the objects themselves have continuous boundaries and we would like to have a smooth path, it is not possible to evaluate the potential at an infinite number of points. A rectangular grid is a simple discretization that places points at the corners of squares (in 2D) or cubes (in 3D). The resolution of the potential and the resulting path increases as the size of the squares decreases. As with all numerical methods there is a tradeoff between computation time (square size) and output accuracy and resolution.

The next section describes the path planning algorithm.

Method Description

Numerical Potential Field Path Planning Tutorial

Matt Greytak
December 9, 2005

Contents:

Part 3: Method Description

Potentials, Force, and Work

A potential is the path integral of a force. If the force is conservative, like gravity or the electrostatic force, then the potential is path-independent. If the force is not conservative, like friction, then the potential at every point depends on the path used to get there. A potential is defined relative to some reference point; we choose the reference point to be the goal point, where the potential is zero. Another way to look at the potential is the work required to move from the goal to any point in the space. Because there are many paths from the goal to any point, we define the potential to be the smallest possible potential, meaning the potential corresponding to the path with the least work. If the potential is computed for the entire space then the optimal path, the one that requires the least amount of work, is found simply by descending the steepest slope of the potential until the goal is reached.

In general forces are vectors, we can express friction-type forces as scalars because the force resists motion regardless of the direction of motion. In a discrete setting the scalar force is the force required to move to a point from any neighboring point. Forces are linear, meaning that the total force is simply the sum of forces from different sources. However, potentials are not linear because they are defined as the minimum over all paths to the point.

Goal Force and Potential

Because forces are linear we can break them up into their various components. First let us focus on the goal force. We want to generate a path from the start to the goal, so we create a potential that is minimum at the goal and grows with distance from the goal. There are many forces that could meet this requirement. The simplest is a constant (flat) goal force like pure friction: the force at every point is 1. The potential for this goal force is simply the distance to the goal. An alternative goal force is conical: the force at every point is the distance to the goal, and the potential is bowl-shaped. The conical goal force punishes paths that are farther from the goal. For consistency in maze conditions a flat goal force is more appropriate.

Obstacle Force

We now know how to make a force and potential that will get us from the start to the goal. However, path planning is never so trivial. The real world has obstacles that the robot must avoid while still getting to the goal as quickly as possible. Just as we created a goal force to bring the robot towards the goal, we can also create obstacle forces to keep the robot away from obstacles. If the force around obstacles is high then the path planner will avoid approaching them because that would mean increasing potential energy. If the potential goes to infinity at the object's boundary then the path will never go too close. A suitable function is the inverse of the distance to the closest boundary of the obstacle. In practice it makes sense to limit the influence of the obstacle to a certain distance. The details of how this is done are explained in the Java implementation description.

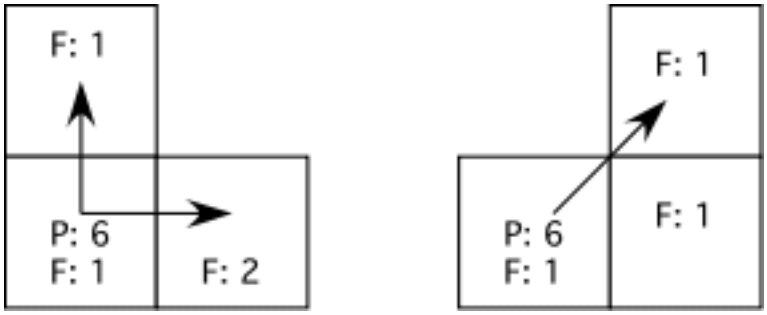
Combining the Forces

The goal and obstacle forces are linear, which means that we can simply add them to get a total force at each location. The total force at a point represents the amount of effort it takes to move to that point from any neighboring point. Far from any obstacles the constant goal force causes all local movements to use the same effort. Therefore the most direct path to the goal is the path that requires the least work (that is, the path that passes through the fewest points). Near obstacles, movements towards the obstacle require more effort than movements away from the obstacle because the force is higher at points closer to the obstacle. Given the choice between a path near an obstacle (passing through points of high force) and a path that avoids the obstacle (passing through points of lower force), the best path is the one that avoids the obstacle because the total work required will be lower.

As stated above, while the forces can be combined linearly, the potentials cannot. This is because the potential at each point depends on the path taken to get there. The potential is not linear, and to evaluate it we must use the total force (goal plus obstacle).

Evaluating the Potential

Now we know how to compute a value for the force at every point in the space. Our next task is to construct the potential in a manner that avoids local minima in which the path could get stuck. We do this by working backwards from the goal, expanding outwards (in space) and upwards (in potential). From each point the neighboring points are examined. The potential at the neighboring point is compared with the sum of the potential at the current point and the force required to get to the new point. If the second value is smaller than the first, then the potential at the neighboring point is replaced with the second value. In this manner the potential at each point becomes the integral of the force along the shortest path from that point to the goal. Such a formulation automatically avoids local minima because the potential always grows along paths away from the goal.



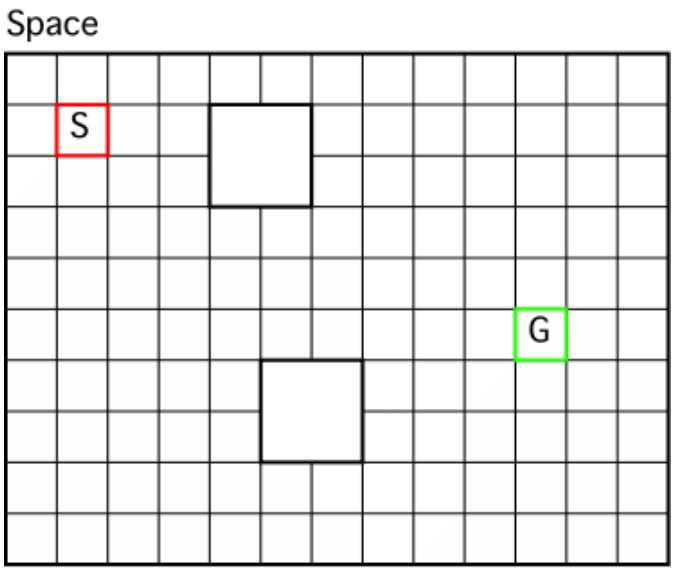
*If the potential of the lower-left square is 6, then the new potential for the square above it will be 7 = 6 + 1, and the new potential for the square to its right will be 8 = 6 + 2. The potential can also be computed along diagonals. In the example on the right the new potential for the upper-right square will be 7.414 = 6 + 1*1.414, where 1.414 is the diagonal distance to the new square. Note that this is a smaller value than if we had to go through the lower-right square, in which case the potential would be 8 = 6 + 1 + 1.*

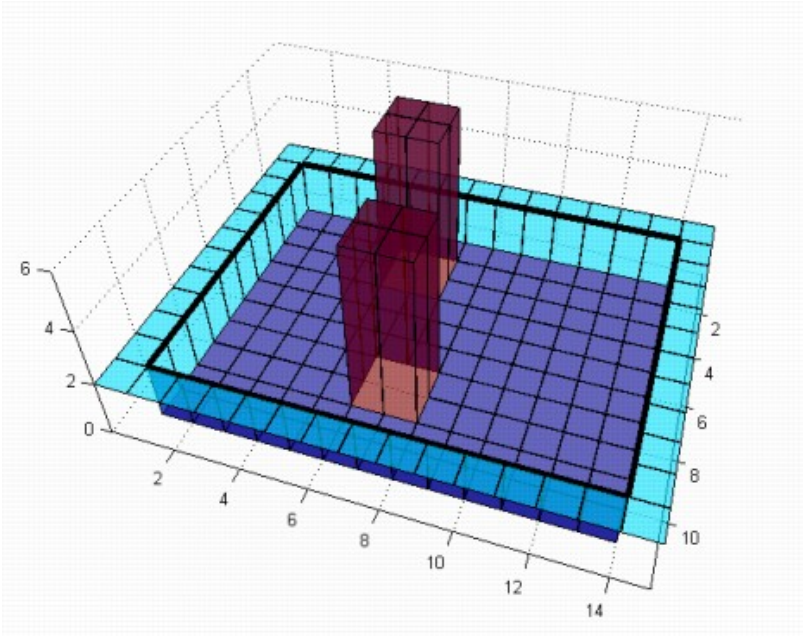
Constructing the Path

At this point we have a potential function for the entire. The path is constructed by starting at the start position and, in each step, moving to the neighboring position with the lowest potential. If two neighboring points have the same lowest potential then it means that either point will get to the goal with the same total work. Ties can be broken randomly or with a convention such as (in 2D) the point farthest to the right and the point farthest down. The path can be constructed with diagonals if all eight surrounding points are examined instead of just the four above, below, to the left, and to the right. However, the diagonal paths may get closer to obstacle corners than desired.

Pseudocode and Example

Suppose that we are trying to find a path from the red square to the green square in the world below. There are two obstacles and a wall surrounding the world.





Left: The input to the path planner: a space with obstacles, a start position, and a stop position. Right: A 3D representation of the space.

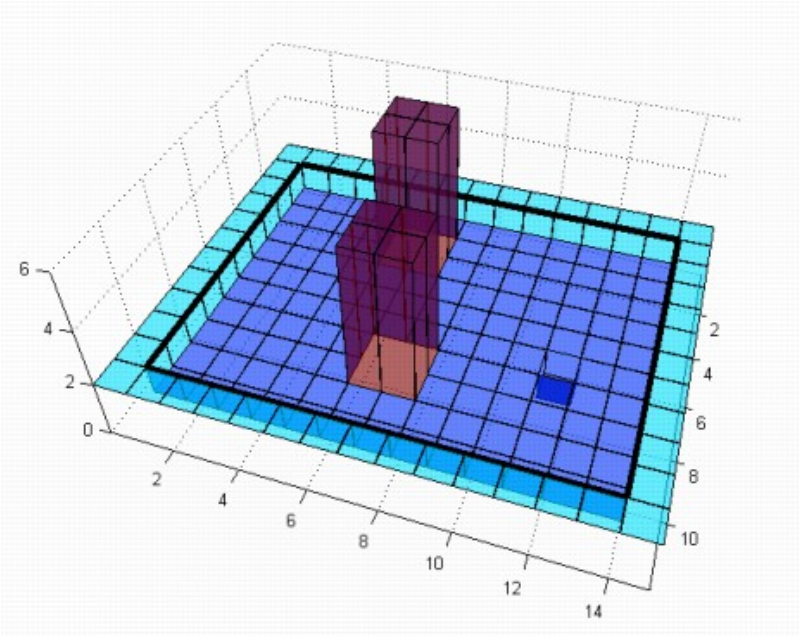
The first step is to create the goal force. If we use a flat goal force then the force at every point except the goal is 1, and the force at the goal is 0.

For all points x in the space, FG(x) = 1
FG(xgoal) = 0

This creates a goal force as shown below:

Goal Force

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



A simple goal force is 1 everywhere but the goal. This is like a friction force; it takes one unit of force to move from any square to any neighboring square.

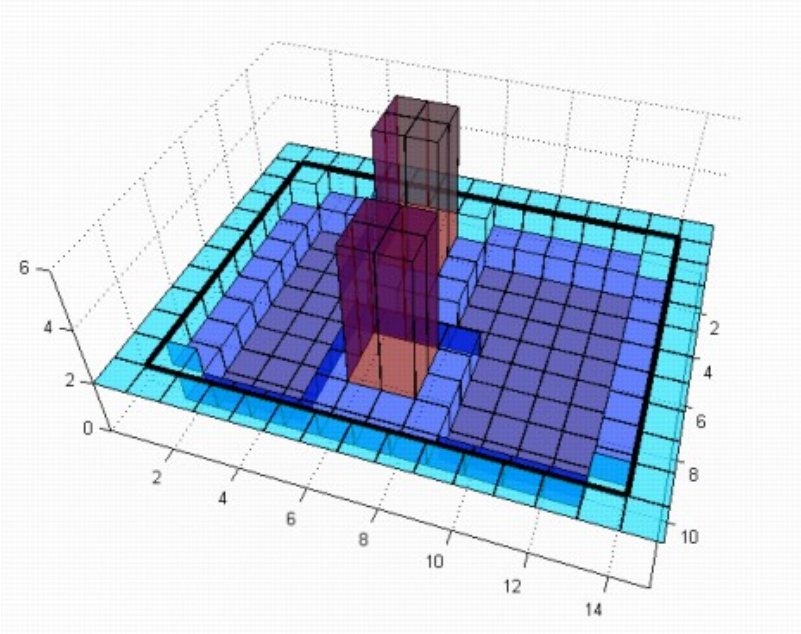
Next we make the obstacle force. For simplicity we will make the obstacle force equal to 1 in the surrounding grid points and 0 everywhere else. Remember that obstacle forces add, so if a point is within one grid point of two obstacles then its force is 2.

For all points x in the space, FO(x) = 0
For all obstacles K,
    For all points x surrounding obstacle k, FO(x) = FO(x) + 1
end obstacle loop

This creates the obstacle force as shown below:

Obstacle Force

2	1	1	2	2	2	2	1	1	1	1	1	2
1	0	0	1			1	0	0	0	0	0	1
1	0	0	1			1	0	0	0	0	0	1
1	0	0	1	1	1	1	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	1	1	1	1	0	0	0	0	1
1	0	0	0	1			1	0	0	0	0	1
1	0	0	0	1			1	0	0	0	0	1
1	0	0	0	1	1	1	1	0	0	0	0	1
2	1	1	1	1	1	1	1	1	1	1	1	2



A simple obstacle force is 1 within one unit of an obstacle (and infinite inside an obstacle). Notice that the walls are treated as obstacles, and obstacle forces add in points that are within one unit of multiple obstacles.

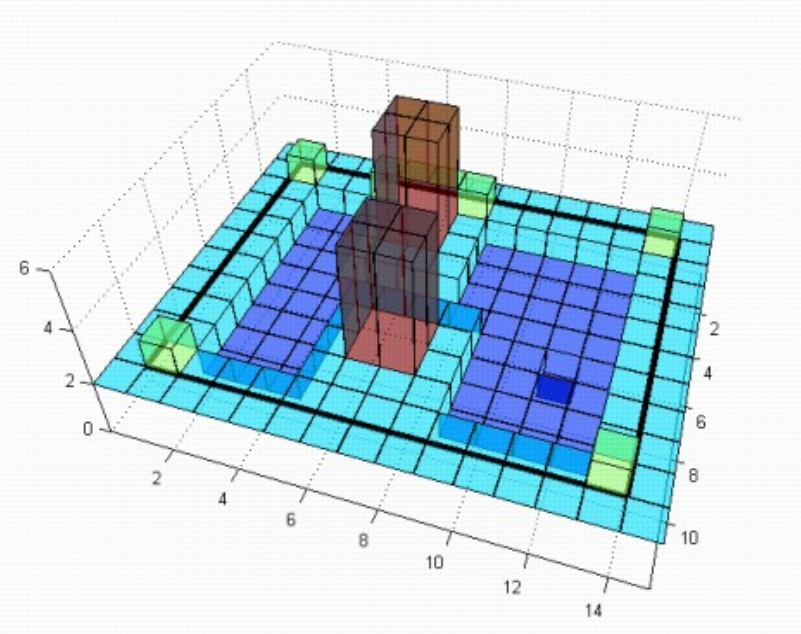
Next sum the goal force and the obstacle force to get the total force.

For all points x in the space, F(x) = FG(x) + FO(x)

The result is shown below:

Total Force

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



The total force is simply the sum of the goal force and the obstacle force.

Now it is time to evaluate the potential at every point. We start by assuming that the potential at every point is infinity (or a very large number). Then, starting with the goal, we find the points that have the lowest potential and look at their neighbors to update their potentials if necessary. We use a queue to

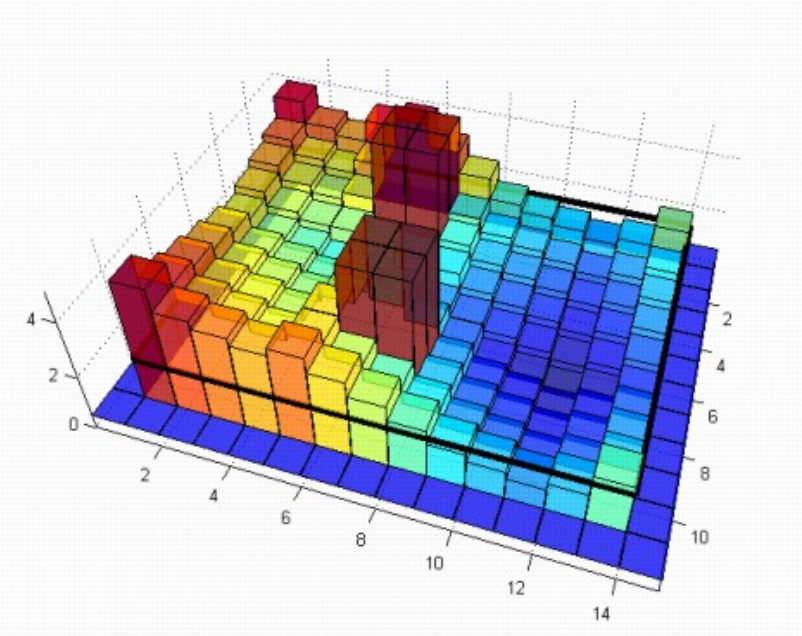
keep track of the points that must be examined.

```
For all points x in the space, U(x) = 10,000
U(xgoal) = 0
Add xgoal to the queue Q
While Q is not empty,
        Remove the point xi with the minimum U from Q
        For all points xj that surround xi,
                If U(xj) > U(xi) + F(xj)
                        U(xj) = U(xi) + F(xj)
                        Add xj to Q
                end if
        end neighbor for loop
end while loop
```

Now all points have a potential value assigned to them, and it looks like this:

Total Potential

18	15	14	17	18	15	12	9	8	7	6	7	10
15	13	12	14			9	7	6	5	4	5	7
14	12	11	12			8	6	5	4	3	4	6
13	11	10	10	9	8	7	5	4	3	2	3	5
12	10	9	8	7	6	5	4	3	2	1	2	4
13	11	10	9	9	8	6	4	2	1	0	1	3
14	12	11	10	11			5	3	2	1	2	4
15	13	12	11	13			6	4	3	2	3	5
16	14	13	12	13	11	9	7	5	4	3	4	6
19	16	15	14	15	13	11	9	7	6	5	6	9



The potential at any point is the minimum amount of force that is picked up by taking any path to the goal. It is computed by starting at the goal and working outwards until every point has the minimum possible potential. Note that the vertical axis of the 3D plot is scaled down.

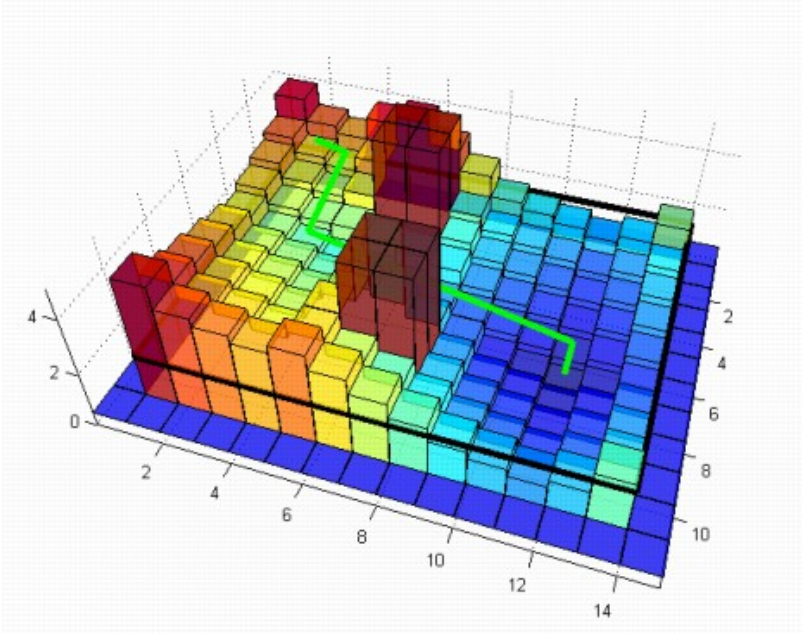
The final step is to make the path from a point xstart.

```
xi = xstart
While xi ≠ xgoal,
        x(i+1) = argmin(for xj neighbors of xi) of U(xj)
end while loop
```

The final path is shown in orange below.

Straight Path

18	15	14	17	18	15	12	9	8	7	6	7	10
15	13	12	14			9	7	6	5	4	5	7
14	12	11	12			8	6	5	4	3	4	6
13	11	10	10	9	8	7	5	4	3	2	3	5
12	10	9	8	7	6	5	4	3	2	1	2	4
13	11	10	9	9	8	6	4	2	1	0	1	3
14	12	11	10	11			5	3	2	1	2	4
15	13	12	11	13			6	4	3	2	3	5
16	14	13	12	13	11	9	7	5	4	3	4	6
19	16	15	14	15	13	11	9	7	6	5	6	9



The path is found by moving to the neighboring square with the lowest potential, starting at any point in the space and stopping when the goal is reached. There are multiple optimal paths, each of which will require the same amount of work; for example, we could start by moving down instead of to the right.

The next section describes how this algorithm is implemented in Java.

[Java Implementation]

Numerical Potential Field Path Planning Tutorial

Matt Greytak
December 9, 2005

Contents:

Part 4: Java Implementation

The Numerical Potential Fields path planning algorithm described in the Method Description section has been implemented in a Java class NPFpathplanner.java. It can be run using the JUnit test architecture; the test program used to run the example is NPFpathplannerTest.java. The required files can be downloaded here as a .zip archive. The *NPFpathplanner* class has the following structure:

```
public class NPFpathplanner {
        void setspace(width, height);
        void drawspace();
        void addobstacle(obstacle);
        void setgoal(goalpoint);
        Path findpath(startpoint);
        int closestobstacle();
        double closestdistanceobs(obstacle);
        double getpathlength();
}
```

To start a problem using this code you first define a rectangular space using the *setspace* method. The width and height are integers that define the discretization of the problem. All squares in the space have an edge length of 1. The space starts with walls along is boundaries, implemented as obstacles. More obstacles can be added to the space using the *addobstacle* method. In this implemenation an obstacle is a *Rectangle2D.Double* object. The obstacle force is updated each time an obstacle is added. The position of the goal is set with the *setgoal* method, where *goalpoint* is a *PotPoint* object (described below). The *findpath* method computes the potential for the entire space and, starting at the *startpoint*, constructs a path to the goal. The path is returned as a *Path* object (described below). Three other methods are used to evaluate the final path: *closestobstacle* returns the obstacle number that the path comes closest to (user-defined obstacles only, the first is number 1), *closestdistanceobs* finds the closest

distance that the path comes to a particular user-defined obstacle, and *getpathlength* returns the total length of the path. Finally, the *drawspace* method makes a plot of the space, the obstacles, the forces, the potential, and the path.

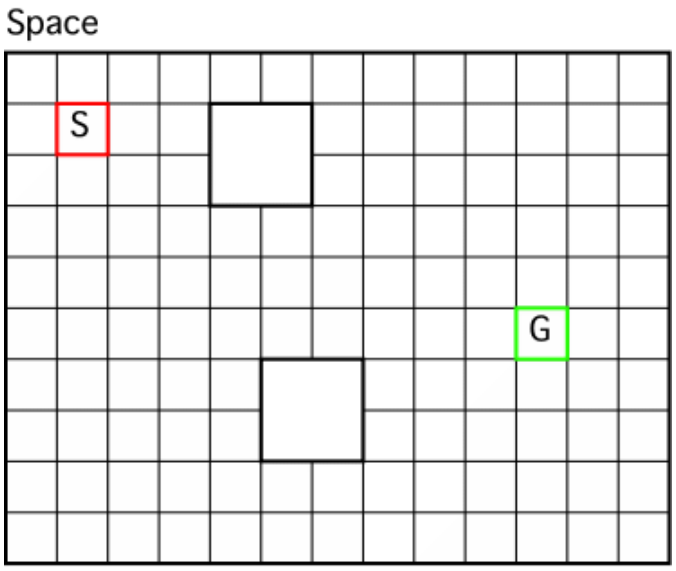
Two custom classes are used by *NPFpathplanner*: PotPoint.java and Path.java.

> *PotPoint* is an extension of *Point2D.Double*, forced to accept only integer values for the postion. The class also contains a constant value to store the potential associated with the point.
>
> *Path* is an extension of *LinkedList* with a constant value to store the path length. The path length is actually computed by the *findpath* method.

There are several options that can be set in the *NPFpathplanner* class; most are self-explanatory.

> boolean *drawgoalforce* - draw the goal force in each square; blue intensity is proportional to force
> boolean *drawobsforce* - draw the obstacle force in each square; red intensity is proportional to force
> boolean *drawpotential* - draw the potential as circles in each square; gray intensity is proportional to potential
> boolean *drawpath* - draw the path as a yellow line from the start to the goal
> boolean *forceflat* - use a flat goal force; if false, use a conical goal force
> boolean *usediagonals* - use diagonals when computing the potential and the path; if false, only look above, below, left, and right
> int *obsinfluence* - set the radius of influence of the obstacles

As a reminder we are trying to find the path from the red square to the green square in the space below:



Problem to solve: find a path from the red square to the green square while avoiding the obstacles.

The following Java commands are found in the NPFpathplannerTest.java JUnit test file. To run the program you can type the following at the command line (in Mac OS X; may vary slightly for other

operating systems):

```
java junit.swingui.TestRunner NPFpathplannerTest
```

To start our example problem we instantiate the class:

```
NPFpathplanner pp = new NPFpathplanner();
```

Next we set the options to match our problem.

```
pp.usediagonals = false;
pp.forceflat = true;
pp.obsinfluence = 1;
```

We use the *setspace* method to define the problem space.

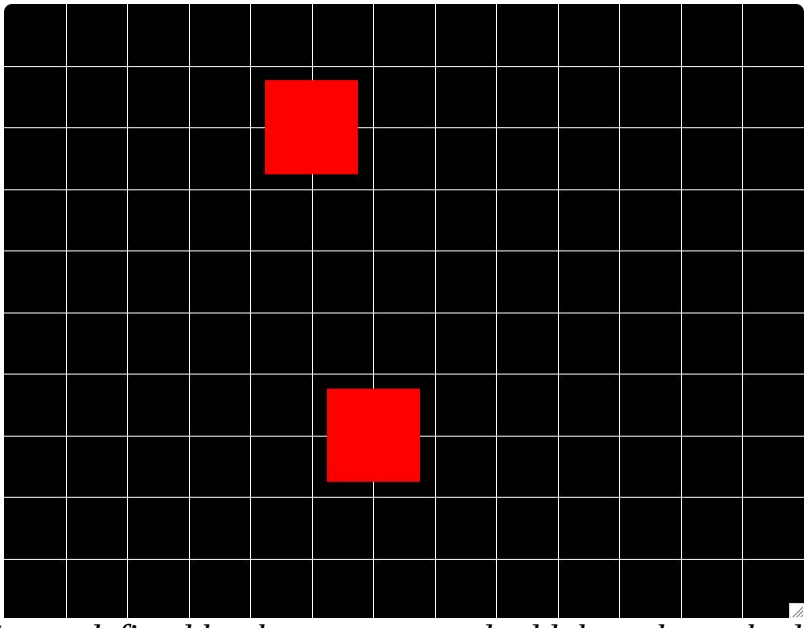
```
pp.setspace(13, 10);
```

The *setspace* method creates a list of points corresponding to every discrete location in the space. It also calls the *addobstacle* method four times to build the four walls.

We then call the *addobstacle* method twice ourselves to put the square obstacles in the interior of the space. The obstacles are *Rectangle2D.Double* objects which are specifed by a horizonal position, a vertical position, a width and a height. The corners of the obstacles are located in the center of the squares, so if an obstacle occupies a single square then its width and height are 0. The obstacles in our example occupy two squares vertically and horizontally, so the width and height are 1.

```
pp.addobstacle(new Rectangle2D.Double(5, 2, 1, 1));
pp.addobstacle(new Rectangle2D.Double(6, 7, 1, 1));
```

Right now the space looks like the following:



Space defined by the setspace and addobstacle methods.

The *addobstacle* method creates the obstacle force by expanding the obstacles by one point at a time up to the size of the obstacle influence. Every time the obstacle is expanded the force of the interior points is increased. The amount of the force increase is larger when the obstacle influence distance is larger.

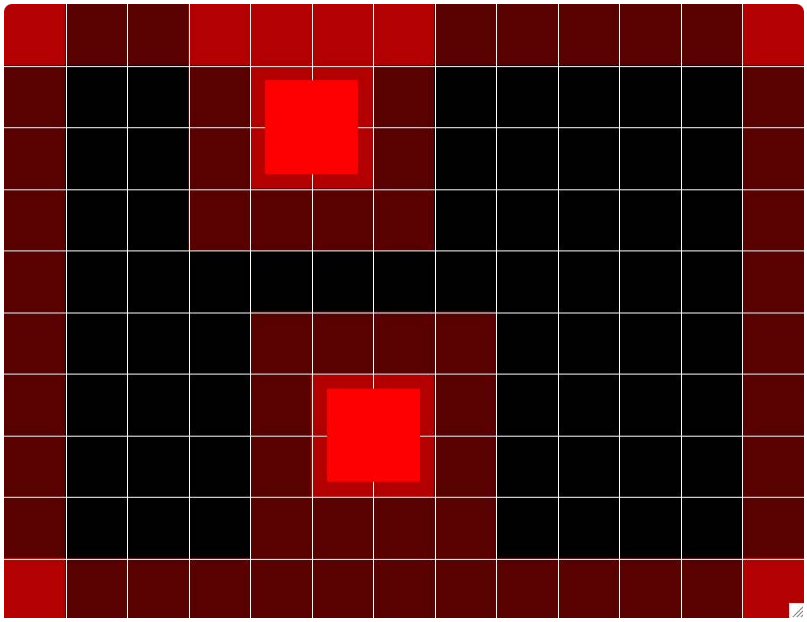
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	1	1	1	0	0
0	0	1		1	0	0
0	0	1	1	1	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

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

1	1	1	1	1	1	1
1	3	3	3	3	3	1
1	3	6	6	6	3	1
1	3	6		6	3	1
1	3	6	6	6	3	1
1	3	3	3	3	3	1
1	1	1	1	1	1	1

The obstacle influence is 1 on the left, 2 in the middle, and 3 on the right. When the obstacle influence is 1, the obstacle is expanded once and the force is increased by 1 at all interior points. When the obstacle influence is 2, the obstacle is expanded twice; the first time the force is increased by 2 and the second time it is increased by 1. When the obstacle influence is 3, the obstacle is expanded three times; the force is increased by 3, then 2, then 1. There are no walls in this example.

The obstacle force for an obstacle influence of 1 is plotted below.

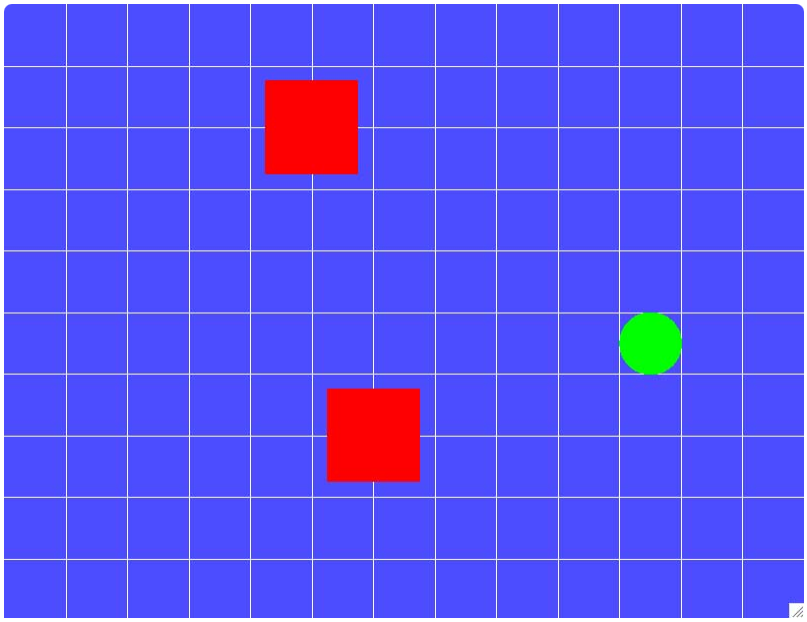


Obstacle force with an influence distance of 1.

The *setgoal* method defines the goal location and creates the goal force.

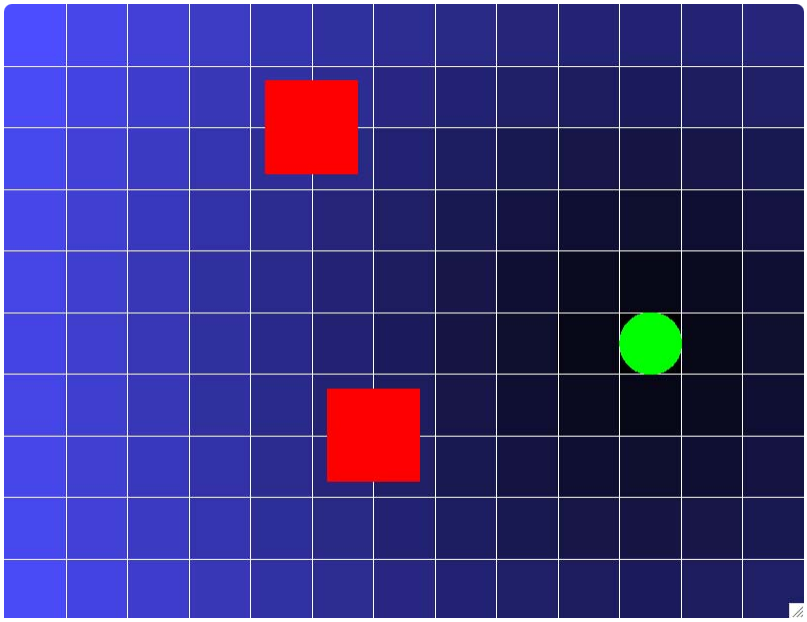
```
pp.setgoal(new PotPoint(10,6));
```

By default a flat goal force is used, which looks like the following:



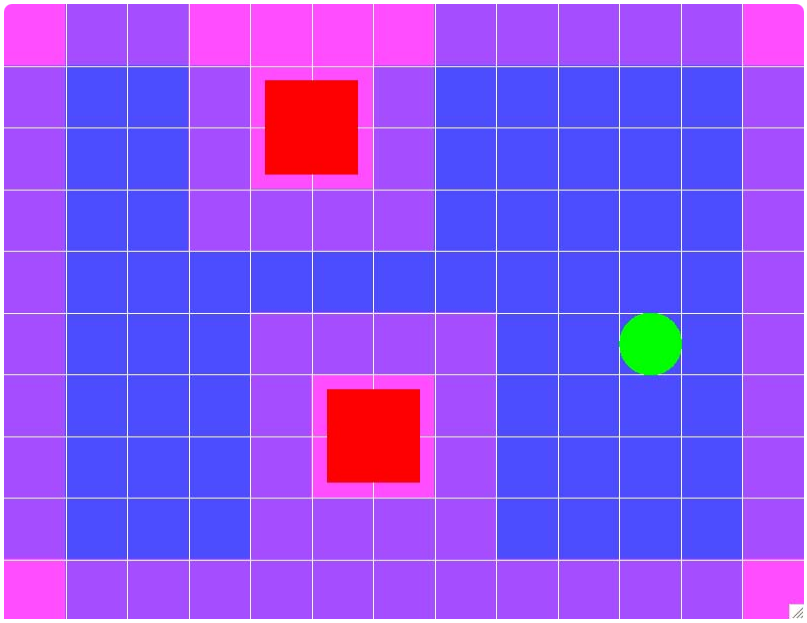
Flat goal force and the goal position.

If we used a conical goal force instead (goal force equals direct distance to the goal), it would look like:



Conical goal force and the goal position.

The total force using a flat goal force is plotted below.

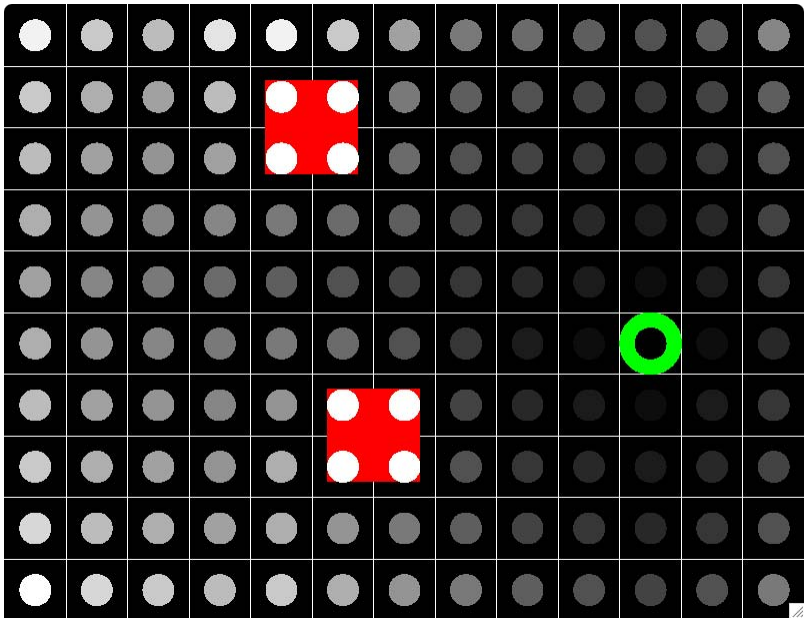


Total force with an obstacle influence of 1 and a flat goal force.

The potential is computed in the same step as the path construction, so next we call the *findpath* method using our start point.

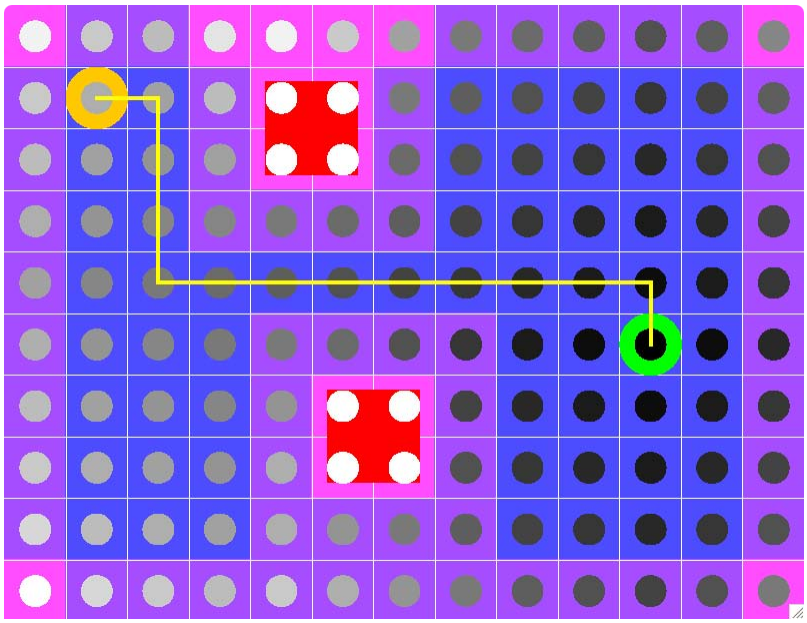
```
Path finalpath = pp.findpath(new PotPoint(2,2));
```

This method creates the potential by expanding outward from the goal until all points have the lowest possible potential. Graphically the potential is represented as circles whose brightness is proportional to potential value. As seen below the potential is highest inside the obstacles and lowest at the goal.



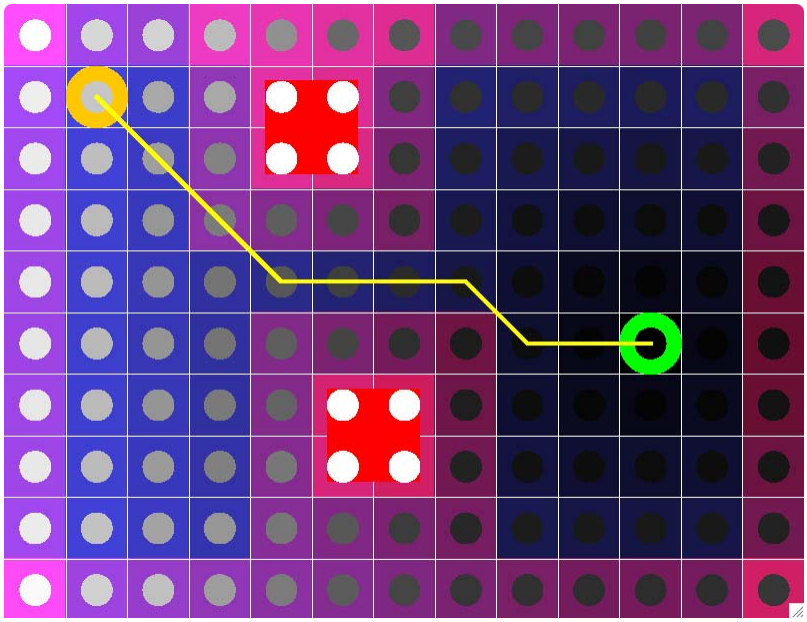
Potential at every point; it is highest in the obstacles and lowest at the goal. Elsewhere it is generally higher farther from the goal and near obstacles.

The path is computed by always moving to the neighbor with the lowest potential, beginning with the start point and ending (necessarily) at the goal point. No potential is lower than the potential at the goal point and all paths lead to the goal so a path is guaranteed to be found.



Path returned by the planner. It moves down the steepest gradient of potential from the start to the goal.

If we use the conical goal force and we use diagonals when evaluating the potential and constructing the path then the result looks like the following.



Result if a conical goal force is used and diagonals are allowed.

The JUnit test program prints the path and path information to the command line. With a flat goal force and no diagonals the output is as follows:

Path Returned:
[2, 2]
[3, 2]
[3, 3]
[3, 4]
[3, 5]
[4, 5]
[5, 5]
[6, 5]
[7, 5]
[8, 5]
[9, 5]
[10, 5]
[11, 5]
[11, 6]

Path Length:
13.0
Closest Obstacle:
1
Closest Distance to Obstacle:
2.0

The next section describes the performance of the algorithm.

Performance

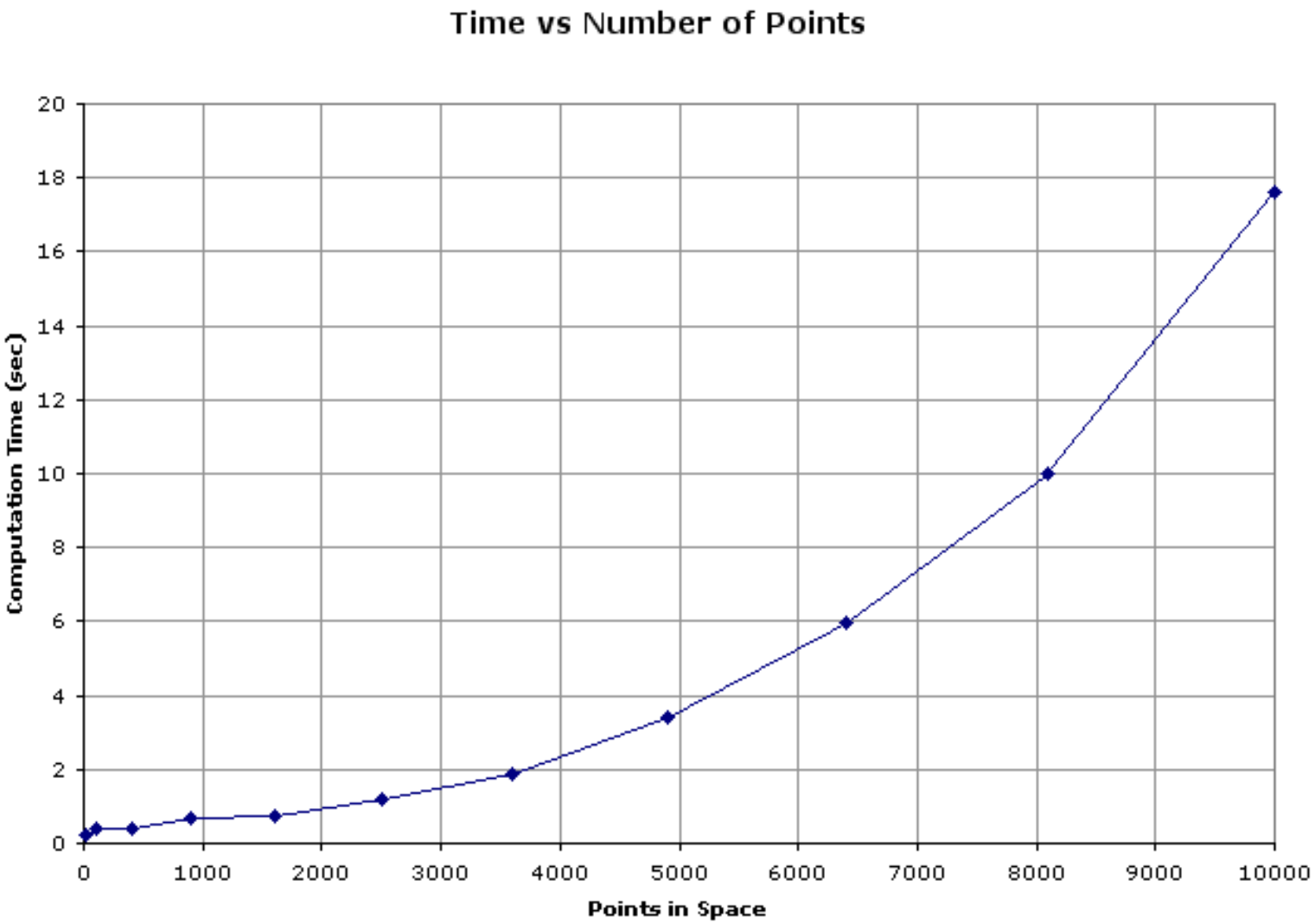
Numerical Potential Field Path Planning Tutorial

Matt Greytak
December 9, 2005

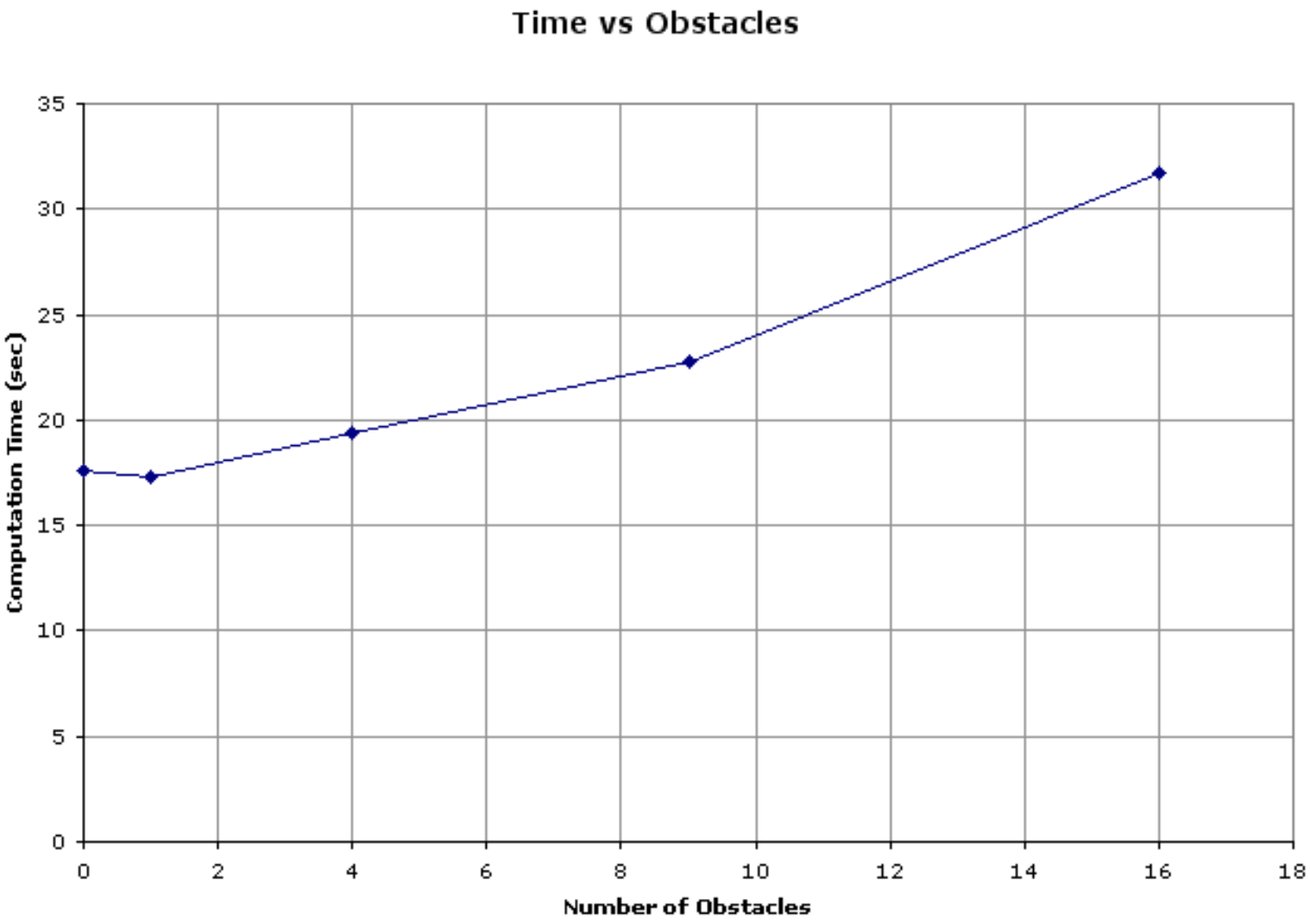
Contents:

Part 5: Performance

To study the performance of the algorithm it was tested on an empty space with a flat goal force and no obstacle influence. The space size was increased from 10 x 10 (100 points total) to 100 x 100 (10,000 points total). The algorithm was run to find the path from the upper left corner to the lower right corner, using diagonals. The time to run the algorithm for each space size is plotted below. It is approximately O(n^2): the queue is accessed n times and, for an unstructured queue, the time to find the point with the lowest potential is O(n).



The algorithm is linear in the number of obstacles because the algorithm looks at each point in the space whenever a new obstacle is added.

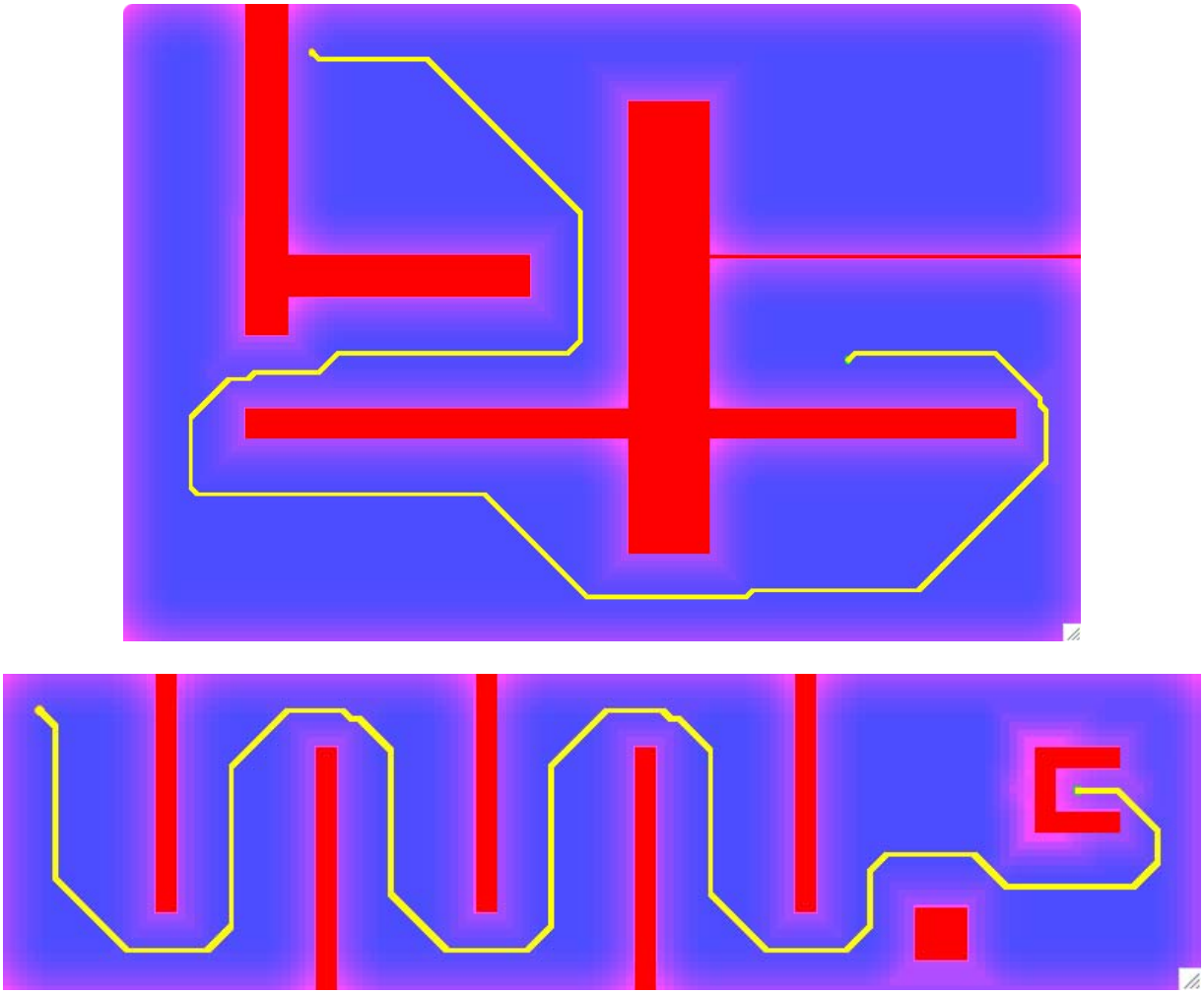


There is a slight savings if neighbors are not examined: 15.8 seconds for the 100 x 100 space, compared with 17.6 seconds if diagonals are used.

Using a conical goal force does not significantly change the computation time: 18.1 seconds, compared with 17.6 seconds if a flat goal force is used.

Obstacle size does not affect computation time because each point in the space is examined when computing the obstacle force regardless.

In general a flat goal force is best for maze-type obstacles as shown below.



The next section includes a Java applet to interactively solve a path planning problem.

Java Applet