Finding the shortest path to the ground while avoiding obstacles

devblogs.microsoft.com/oldnewthing/20140728-00

July 28, 2014

Raymond Chen

Today's Little Program solves the following problem:

> Consider a two-dimensional board, tall and narrow. Into the board are nailed a number of horizontal obstacles. Place a water faucet at the top of the board and turn it on. The water will dribble down, and when it hits an obstacle, some of the water will go left and some will go right. The goal is to find the shortest path to the ground from a given starting position, counting both horizontal and vertical distance traveled.

In the above diagram, the water falls three units of distance until it encounters Obstacle 1, at which some goes to the left and some goes to the right. The water that goes to the left travels three units of distance before it reaches the end of the obstacle, then falls three units and encounters Obstacle 2. Upon reaching Obstable 2, the water can again choose to flow either left or right. The water that flows to the left falls to the ground; the water that flows to the right falls and encounters a third obstacle. From the third obstacle, the water can flow left or right, and either way it goes, it falls to the ground. On the other hand, the water that chose to flow to the right when it encountered Obstable 1 iwould fall past Obstacle 2 (which is not in a position to intercept the water) and land directly on Obstacle 3.

In the above scenario, there are five paths to the ground.

- From Obstacle 1, flow left, then from Obstacle 2, flow left again. Total distance traveled: 17 units.
- From Obstacle 1, flow left, then from Obstacle 2, flow right, then from Obstacle 3, flow left. Total distance traveled: 18 units.
- From Obstacle 1, flow left, then from Obstacle 2, flow right, then from Obstacle 3, flow right. Total distance traveled: 20 units.
- From Obstacle 1, flow right, then from Obstacle 3, flow left. Total distance traveled: 16 units.
- From Obstacle 1, flow right, then from Obstacle 3, flow right. Total distance traveled: 14 units.

In this case, the shortest path to the ground is the last path.

There are many ways to attack this problem. The brute force solution would be to enumerate all the possible paths to the ground, then pick the shortest one.

A more clever solution would use a path-finding algorithm like A*, where the altitude above the ground is the heuristic.

In both cases, you can add an optimization where once you discover two paths to the same point, you throw out the longer one. This may short-circuit future computations.

But I'm going to use an incremental solution, since it has the advantage of incorporating the optimization as a convenient side-effect. Instead of studying individual drops of water, I'm going to study all of them at once. At each step in the algorithm, the data structures represent a horizontal cross-section of the above diagram, representing all possible droplet positions at a fixed altitude.

In addition to collapsing redundant paths automatically, this algorithm has the nice property that it can be done as an on-line algorithm: You don't need to provide all the obstacles in advance, as long as the obstacles are provided in order of decreasing altitude.

Instead of presenting the raw code and discussing it later (as is my wont), I'll explain the code as we go via code comments. We'll see how well that works.

I originally wrote the program in C# because I thought I would need one of the fancy collection classes provided by the BCL, but it turns out that I didn't need anything fancier than a hash table. After I wrote the original C# version, I translated it to JavaScript, which is what I present here.

<!–

```
// Oh you're so clever - you are doing a view.source to see the C# version.
// You can put your finger by the side of your nose as a signal.
// http://ask.metafilter.com/4447/
using System;
using System.Collections.Generic;
using System.Linq;
// An Obstacle describes the position of a single obstacle
class Obstacle
{
 public double Left { get; set; }
 public double Right { get; set; }
 public double Y { get; set; }
}
// A Step describes the last point in a path,
// the cost to get there, and
// a reference to the rest of the path.
class Step
{
 public double X { get; set; }
 public double Y { get; set; }
 public double Cost { get; set; }
 public Step Previous { get; set; }
 // Add a step to an existing step
 public Step To(double x, double y, double incrementalCost)
 {
  return new Step { Previous = this, X = x, Y = y,
                    Cost = Cost + incrementalCost };
 }
}
// Locations tracks all the possible droplet locations at a particular
// altitude. The positions are indexed by X-coordinate for easy look-up.
class Locations : SortedList<double, Step>
{
 // Record a droplet position
 public void Add(Step step)
 {
  // If no previous droplet at this position or the new droplet
  // has a cheaper path, then remember this droplet.
  if (!this.ContainsKey(step.X) || step.Cost < this[step.X].Cost)
  {
   this[step.X] = step;
  }
 }
}
static class Extensions
{
 // Like Min, but returns the item with the minimum value rather than the
 // minimum value itself.
 public static T MinIndirect<T, V>(
    this IEnumerable<T> source,
    Func<T, V> func) where V : IComparable
 {
```

```
    return source.Aggregate(default(T), (sofar, next) =>
       (sofar == null || func(next).CompareTo(func(sofar)) < 0 ?
                                                    next : sofar));
 }
}
class Program
{
 // Take an existing collection of locations and updates them to account
 // for a new obstacle. Obstacles must be added in decreasing altitude.
 // (Consecutive duplicate altitudes allowed.)
 static Locations FallTo(Locations oldLocations, Obstacle obstacle)
 {
  Locations newLocations = {};
  foreach (var pair in oldLocations)
  {
   var step = pair.Value;
   var dy = step.Y - obstacle.Y; // distance fallen
   if (dy > 0)
   {
    // fall to the obstacle's altitude
    step = step.To(step.X, obstacle.Y, dy);
   }
   // If the falling object does not hit the obstacle,
   // then there is no horizontal displacement.
   if (step.X <= obstacle.Left || step.X >= obstacle.Right)
   {
    newLocations.Add(step);
   }
   else
   {
    // The falling object hit the obstacle.
    // Split into two droplets, one that goes left
    // and one that goes right.
    newLocations.Add(step.To(obstacle.Left, obstacle.Y,
                             step.X - obstacle.Left));
    newLocations.Add(step.To(obstacle.Right, obstacle.Y,
                             obstacle.Right - step.X));
   }
  }
  return newLocations;
 }
 static void PrintPath(Step firstStep)
 {
  System.Console.Write("Cost = {0}: ", firstStep.Cost);
  // Walk the path backwards, then reverse it so we can print
  // the results forward.
  var path = new List<Step>();
  for (var step = firstStep; step != null; step = step.Previous)
  {
   path.Add(step);
  }
  path.Reverse();
```

```
  foreach (var step in path)
  {
   System.Console.Write("({0},{1}) ", step.X, step.Y);
  }
  System.Console.WriteLine();
 }
 // Debugging function
 static void PrintLocations(Locations l)
 {
  foreach (var pair in l)
  {
   PrintPath(pair.Value);
  }
 }
 static void Main()
 {
  var l = {};
  System.Console.Write("Initial X position: ");
  var x = double.Parse(System.Console.ReadLine());
  System.Console.Write("Initial Y position: ");
  var y = double.Parse(System.Console.ReadLine());
  l[x] = new Step { X = x, Y = y };
  for (;;)
  {
   PrintLocations(l);
   System.Console.Write("Obstacle height (0 = floor): ");
   y = double.Parse(System.Console.ReadLine());
   if (y == 0) break;
   System.Console.Write("Obstacle Left edge: ");
   var left = double.Parse(System.Console.ReadLine());
   System.Console.Write("Obstacle Right edge: ");
   var right = double.Parse(System.Console.ReadLine());
   l = FallTo(l, new Obstacle { Left = left, Right = right, Y = y });
  }
  // Find the cheapest step.
  var best = l.Values.MinIndirect(s => s.Cost);
  // Add a segment from the last obstacle to the floor and print it.
  PrintPath(best.To(best.X, 0, best.Y));
 }
}

—>
```

The inputs which correspond to the diagram above are

- Initial X position = 6, Initial Y position = 12
- Obstacle: Left = 3, Right = 7, height = 9
- Obstacle: Left = 1, Right = 5, height = 6
- Obstacle: Left = 4, Right = 8, height = 3

And here’s the program.

```
function Obstacle(left, right, y) {
 this.left = left;
 this.right = right;
 this.y = y;
}
// A single step in a path, representing the cost to reach that point.
function Step(x, y, cost) {
 this.x = x;
 this.y = y;
 this.cost = cost;
}
 // Add a step to an existing step
Step.prototype.to = function to(x, y) {
 var dx = Math.abs(this.x - x);
 var dy = Math.abs(this.y - y);
 return new Step(x, y, this.cost + dx + dy);
}
// Record a droplet position
function addDroplet(l, step) {
 // If no previous droplet at this position or the new droplet
 // has a cheaper path, then remember this droplet.
 var existingStep = l[step.x];
 if (!existingStep || step.cost < existingStep.cost) {
  l[step.x] = step;
 }
}
// Take an existing collection of locations and updates them to account
// for a new obstacle. Obstacles must be added in decreasing altitude.
// (Consecutive duplicate altitudes allowed.)
function fallTo(oldLocations, obstacle) {
 var newLocations = {};
 for (var x in oldLocations) {
  var step = oldLocations[x];
  // fall to the obstacle's altitude
  step = step.to(step.x, obstacle.y);
  // If the falling object does not hit the obstacle,
  // then there is no horizontal displacement.
  if (step.x <= obstacle.left || step.x >= obstacle.right) {
   addDroplet(newLocations, step);
  } else {
   // The falling object hit the obstacle.
   // Split into two droplets, one that goes left
   // and one that goes right.
   addDroplet(newLocations, step.to(obstacle.left, obstacle.y));
   addDroplet(newLocations, step.to(obstacle.right, obstacle.y));
  }
 }
 return newLocations;
}
function printStep(step) {
 console.log("Cost = " + step.cost + ": " + step.x + "," + step.y);
}
```

```
// Debugging function
function printLocations(l) {
 for (var x in l) printStep(l[x]);
}
function shortestPath(x, y, obstacles) {
 var l = {};
 l[x] = new Step(x, y, 0);
 printLocations(l);
 obstacles.forEach(function (obstacle) {
  l = fallTo(l, obstacle);
  console.log(["after", obstacle.left, obstacle.right, obstacle.y].join(" "));
  printLocations(l);
  console.log("===");
 });
 // Find the cheapest step.
 var best;
 for (x in l) {
  if (!best || l[x].cost < best.cost) best = l[x];
 }
 // Fall to the floor and print the result.
 printStep(best.to(best.x, 0));
}
shortestPath(6,12,[new Obstacle(3,7,9),
                   new Obstacle(1,5,6),
                   new Obstacle(4,8,3)]);
```

This program finds the cost of the cheapest path to the floor, but it merely tells you the cost and not how the cost was determined. To include the winning path, we need to record the history of how the cost was determined. This is a standard technique in dynamic programming: In addition to remembering the best solution so far, you also remember how that solution was arrived at by remembering the previous step in the solution. You can then walk backward through all the previous steps to recover the full path.

```
// A single step in a path, representing the cost to reach that point
// and the previous step in the path.
function Step(x, y, cost, previous) {
 this.x = x;
 this.y = y;
 this.cost = cost;
 this.previous = previous;
}
 // Add a step to an existing step
Step.prototype.to = function to(x, y) {
 var dx = Math.abs(this.x - x);
 var dy = Math.abs(this.y - y);
 // These next two test are not strictly necessary. They are for style points.
 if (dx == 0 && dy == 0) {
  // no movement
  return this;
 } else if (dx == 0 && this.previous && this.previous.x == x) {
  // collapse consecutive vertical movements into one
  return new Step(x, y, this.cost + dx + dy, this.previous);
 } else {
  return new Step(x, y, this.cost + dx + dy, this);
 }
}
function printStep(firstStep) {
 // Walk the path backwards, then reverse it so we can print
 // the results forward.
 var path = [];
 for (var step = firstStep; step; step = step.previous) {
  path.push("(" + step.x + "," + step.y + ")");
 }
 path.reverse();
 console.log("Cost = " + firstStep.cost + ": " + path.join(" "));
}
```

Notice that we didn't change any of the program logic. All we did was improve our record-keeping so that the final result prints the full path from the starting point to the ending point.

Raymond Chen

Follow