```
#pragma once
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
#include <map>
#include <set>
#include <math.h>

using namespace std;

# define M_PI  3.14159265358979323846  /* pi */

class DiscreteDistribution
{
	struct ProbabilityNode
	{
		double cost;
		double probability;

		ProbabilityNode() {}

		ProbabilityNode(double x, double prob)
			: cost(x), probability(prob) {}

		bool operator<(const ProbabilityNode& node) const
		{
			return this->cost < node.cost;
		}

		bool operator>(const ProbabilityNode& node) const
		{
			return this->cost > node.cost;
		}

		bool operator==(const ProbabilityNode& node)
		{
			return (this->cost == node.cost) && (this-
>probability == node.probability);
		}

		bool operator!=(const ProbabilityNode& node)
		{
			return !(*this == node);
		}
	};

	struct ProbabilityPair
	{
		ProbabilityNode first;
		ProbabilityNode second;
		ProbabilityPair* left;
		ProbabilityPair* right;

		ProbabilityPair(ProbabilityNode lower,
```

```
ProbabilityNode upper)
                        : first(lower), second(upper), left(NULL),
right(NULL) {}
        };

        struct CompareDistance
        {
                bool operator()(ProbabilityPair* p1,
ProbabilityPair* p2)
                {
                        return (p1->second.cost - p1->first.cost) >
(p2->second.cost - p2->first.cost);
                }
        };

        set<ProbabilityNode> distribution;
        int maxSamples;
        double var;

        double probabilityDensityFunction(double x, double mu,
double var)
        {
                return ((1 / sqrt(2 * M_PI * var)) * exp(-(pow(x -
mu, 2) / (2 * var))));
        }

        void resize(map<double, double>& distroMap)
        {
                // Maybe we don't need to merge any buckets...
                if (distroMap.size() <= maxSamples)
                {
                        return;
                }

                // Gotta merge some buckets...
                priority_queue<ProbabilityPair*,
vector<ProbabilityPair*>, CompareDistance> heap;

                // Groups probabilities into adjacent pairs and
does some pointer assignment for tracking merges
                int cnt = 0;
                ProbabilityNode lastNode;
                ProbabilityPair* lastPair = NULL;
                for (map<double, double>::iterator it =
distroMap.begin(); it != distroMap.end(); it++)
                {
                        ProbabilityNode n(it->first, it->second);

                        if (cnt == 0)
                        {
                                cnt++;
                                lastNode = n;
                                continue;
                        }
```

```
                        ProbabilityPair* p = new
ProbabilityPair(lastNode, n);
                        heap.push(p);

                        p->left = lastPair;
                        p->right = NULL;
                        if (lastPair)
                                lastPair->right = p;
                        lastPair = p;
                        lastNode = n;
                        cnt++;
                }

                // Now, while we still have too many samples, and
the heap isn't empty, merge buckets
                while (distroMap.size() > maxSamples && !
heap.empty())
                {
                        // Get the pair with the lowest distance
between buckets
                        ProbabilityPair* merge = heap.top();
                        heap.pop();

                        // Calculate the new probability and X of
the merged bucket
                        double newProb = merge->first.probability +
merge->second.probability;
                        double newX = (merge->first.probability /
newProb) * merge->first.cost + (merge->second.probability / newProb)
* merge->second.cost;

                        ProbabilityNode newNode(newX, newProb);

                        // Either add this probability to the
existing bucket or make a new bucket for it
                        distroMap[newX] += newProb;

                        // Remove the old probabilities
                        distroMap.erase(merge->first.cost);
                        distroMap.erase(merge->second.cost);

                        // If merge has a pair on its left, update
it
                        if (merge->left)
                        {
                                merge->left->second = newNode;
                                merge->left->right = merge->right;
                        }
                        // If merge has a pair on its right, update
it
                        if (merge->right)
                        {
                                merge->right->first = newNode;
                                merge->right->left = merge->left;
                        }
```

```
                    // Delete the merged pair
                    delete merge;
            }

            // Delete everything on the heap
            while (!heap.empty())
            {
                    ProbabilityPair* p = heap.top();
                    heap.pop();
                    delete p;
            }

            // If we still have too many samples, do it again
            if (distroMap.size() > maxSamples)
                    resize(distroMap);
    }

public:

    DiscreteDistribution() {}
    DiscreteDistribution(int maxSamples) :
maxSamples(maxSamples) {}
    DiscreteDistribution(int maxSamples, double f, double mean,
double d, double error)
            : maxSamples(maxSamples)
    {
            // This is a goal node, belief is a spike at true
value
            if (d == 0)
            {
                    distribution.insert(ProbabilityNode(mean,
1.0));
                    return;
            }

            double stdDev = error / 2.0;
            var = pow(stdDev, 2);

            // Create a Discrete Distribution from a gaussian
            double lower = f;
            double upper = mean + 3 * stdDev;

            double sampleStepSize = (upper - lower) /
maxSamples;

            double currentX = lower;

            double probSum = 0.0;

            vector<ProbabilityNode> tmp;

            // Take the samples and build the discrete
distribution
            for (int i = 0; i < maxSamples; i++)
```

```
		{
			// Get the probability for this x value
			double prob =
probabilityDensityFunction(currentX, mean, var);

			// So if this a goal node, we know the cost
			if (std::isnan(prob) && stdDev == 0)
				prob = 1.0;

			probSum += prob;

			ProbabilityNode node(currentX, prob);

			tmp.push_back(node);

			currentX += sampleStepSize;
		}

		// Normalize the distribution probabilities
		for (ProbabilityNode& n : tmp)
		{
			if (probSum > 0.0 && n.probability != 1.0)
				n.probability = n.probability /
probSum;
			distribution.insert(n);
		}
	}

	// Creates a discrete distribution based on Pemberton's
belief distribution, a uniform between 0 and 1, offset by some g-
value
	DiscreteDistribution(int maxSamples, double g, double d)
		: maxSamples(maxSamples)
	{
		// This is a goal node, belief is a spike at true
value
		if (d == 0)
		{
			distribution.insert(ProbabilityNode(g,
1.0));
			return;
		}

		// Create a Discrete Distribution from a gaussian
		double lower = g;
		double upper = 1.0 + g;

		double sampleStepSize = (upper - lower) /
maxSamples;

		double currentX = lower;

		double sum = 0.0;
		vector<ProbabilityNode> tmp;
```

```
                // Take the samples and build the discrete 
distribution
                for (int i = 0; i < maxSamples; i++)
                {
                        sum += 2 * (1 + g - currentX);
                        ProbabilityNode node(currentX, 2 * (1 + g - 
currentX));

                        tmp.push_back(node);

                        currentX += sampleStepSize;
                }

                // Normalize the distribution probabilities
                for (ProbabilityNode& n : tmp)
                {
                        n.probability = n.probability / sum;
                        distribution.insert(n);
                }
        }

        // Creates a discrete distribution based on Pemberton's 
belief distribution, a uniform between 0 and 1, offset by some g-
value
        DiscreteDistribution(int maxSamples, double g, double d, 
int bf)
                : maxSamples(maxSamples)
        {
                vector<DiscreteDistribution> uniforms;

                for (int i = 0; i < bf; i++)
                {
                        DiscreteDistribution u(maxSamples);
                        uniforms.push_back(u);
                }

                // This is a goal node, belief is a spike at true 
value
                if (d == 0)
                {
                        distribution.insert(ProbabilityNode(g, 
1.0));
                        return;
                }

                // Leaf nodes in this case are a convolution of bf 
uniform distributions between 0 and 1
                double lower = 0.0;
                double upper = 1.0;

                double sampleStepSize = (upper - lower) / 
maxSamples;

                double currentX = lower;
```

```
		for (int i = 0; i < maxSamples; i++)
		{
			// Shift the uniform distros by the leaf's
g-value
			ProbabilityNode node(currentX + g,
sampleStepSize);

			for (DiscreteDistribution& uniform :
uniforms)
				uniform.distribution.insert(node);

			currentX += sampleStepSize;
		}

		// Now convolute the uniform distributions
		for (int i = 1; i < uniforms.size(); i++)
		{
			uniforms[0] = uniforms[0] * uniforms[i];
		}

		this->distribution = uniforms[0].distribution;
	}

	// Creates a delta spike belief
	DiscreteDistribution(int maxSamples, double deltaSpikeValue)
		: maxSamples(maxSamples)
	{

distribution.insert(ProbabilityNode(deltaSpikeValue, 1.0));
	}

	void createFromUniform(int maxSamples, double g, double d)
	{
		this->maxSamples = maxSamples;
		// Clear existing distro
		distribution.clear();

		// This is a goal node, belief is a spike at true
value
		if (d == 0)
		{
			distribution.insert(ProbabilityNode(g,
1.0));
			return;
		}

		// Create a Discrete Distribution from a gaussian
		double lower = g;
		double upper = 1.0 + g;

		double sampleStepSize = (upper - lower) /
maxSamples;

		double currentX = lower;
```

```
		double probStep = 1.0 / maxSamples;

		// Take the samples and build the discrete
distribution
		for (int i = 0; i < maxSamples; i++)
		{
			ProbabilityNode node(currentX, probStep);

			distribution.insert(node);

			currentX += sampleStepSize;
		}
	}

	void createFromGaussian(double f, double mean, double d,
double error)
	{
		// Clear existing distro
		distribution.clear();

		// This is a goal node, belief is a spike at true
value
		if (d == 0)
		{
			distribution.insert(ProbabilityNode(mean,
1.0));
			return;
		}

		double stdDev = error / 2.0;
		var = pow(stdDev, 2);

		// Create a Discrete Distribution from a gaussian
		double lower = f;
		double upper = mean + 3 * stdDev;

		double sampleStepSize = (upper - lower) /
maxSamples;

		double currentX = lower;

		double probSum = 0.0;

		vector<ProbabilityNode> tmp;

		// Take the samples and build the discrete
distribution
		for (int i = 0; i < maxSamples; i++)
		{
			// Get the probability for this x value
			double prob =
probabilityDensityFunction(currentX, mean, var);

			// So if this a goal node, we know the cost
			if (std::isnan(prob) && stdDev == 0)
```

```
                                prob = 1.0;

                        probSum += prob;

                        ProbabilityNode node(currentX, prob);

                        tmp.push_back(node);

                        currentX += sampleStepSize;
                }

                // Normalize the distribution probabilities
                for (ProbabilityNode& n : tmp)
                {
                        n.probability = n.probability / probSum;
                        distribution.insert(n);
                }
        }

        double expectedCost()
        {
                double E = 0.0;

                for (ProbabilityNode n : distribution)
                {
                                E += n.cost * n.probability;
                }

                return E;
        }

        DiscreteDistribution& operator=(const DiscreteDistribution&
rhs)
        {
                if (&rhs == this)
                {
                        return *this;
                }

                distribution.clear();

                distribution = rhs.distribution;
                maxSamples = rhs.maxSamples;

                return *this;
        }

        DiscreteDistribution operator*(const DiscreteDistribution&
rhs)
        {
                DiscreteDistribution csernaDistro(min(maxSamples,
rhs.maxSamples));

                map<double, double> results;
```

```
		for (ProbabilityNode n1 : distribution)
		{
			for (ProbabilityNode n2 : rhs.distribution)
			{
				double probability =
(n1.probability * n2.probability);

				// Don't add to the distribution
if the probability of this cost is 0
				if (probability > 0)
					results[min(n1.cost,
n2.cost)] += probability;
			}
		}

		csernaDistro.resize(results);

		for (map<double, double>::iterator it =
results.begin(); it != results.end(); it++)
		{

csernaDistro.distribution.insert(ProbabilityNode(it->first, it-
>second));
		}

		/*
		cout << csernaDistro.expectedCost() << endl;
		double cdf;
		cout << "Path Cost Node 1,Probability Node 1,CDF
Node 1" << endl;
		cdf = 0.0;
		for (ProbabilityNode n1 : distribution)
		{
			cdf += n1.probability;
			cout << n1.cost << "," << n1.probability <<
"," << cdf << endl;
		}
		cout << endl << endl;
		cout << "Path Cost Node 2,Probability Node 2,CDF
Node 2" << endl;
		cdf = 0.0;
		for (ProbabilityNode n1 : rhs.distribution)
		{
			cdf += n1.probability;
			cout << n1.cost << "," << n1.probability <<
"," << cdf << endl;
		}
		cout << endl << endl;
		cout << "Path Cost Cserna,Probability Cserna,CDF
Cserna" << endl;
		cdf = 0.0;
		for (ProbabilityNode n1 : csernaDistro.distribution)
		{
			cdf += n1.probability;
			cout << n1.cost << "," << n1.probability <<
```

```
"," << cdf << endl;
                }
                cout << endl << endl;
                exit(1);
                */

                return csernaDistro;
        }

        DiscreteDistribution& squish(double factor)
        {
                set<ProbabilityNode> newDistribution;
                double mean = expectedCost();

                // If the squish factor is 1, all values in
distribution will be moved to the mean.
                if (factor == 1)
                {

newDistribution.insert(ProbabilityNode(mean, 1.0));
                        distribution.clear();
                        distribution = newDistribution;

                        return *this;
                }

                /*
                cout << "Before Squish Cost,Before Squish
Probability" << endl;
                for (ProbabilityNode n : distribution)
                {
                        cout << n.cost << "," << n.probability <<
endl;
                }
                cout << endl;
                */

                for (ProbabilityNode n : distribution)
                {
                        double distanceToMean = abs(n.cost - mean);
                        double distanceToShift = distanceToMean *
factor;

                        double shiftedCost = n.cost;

                        if (shiftedCost > mean)
                                shiftedCost -= distanceToShift;
                        else if (shiftedCost < mean)
                                shiftedCost += distanceToShift;


newDistribution.insert(ProbabilityNode(shiftedCost, n.probability));
                }

                distribution.clear();
```

```
		distribution = newDistribution;

		/*
		cout << "Squish Cost,Squish Probability" << endl;
		for (ProbabilityNode n : distribution)
		{
			cout << n.cost << "," << n.probability << 
endl;
		}
		cout << endl;
		*/

		return *this;
	}

	set<ProbabilityNode>::iterator begin()
	{
		return distribution.begin();
	}

	set<ProbabilityNode>::iterator end()
	{
		return distribution.end();
	}
};
```