Simulated Annealing

Background

Simulated annealing is a procedure whereby a system can be optimized by starting with a randomly generated solution \( S_0 \), identifying random changes (there are many different ways the changes can be chosen), and moving to them with some probability \( P(S_{\text{curr}}, S_{\text{next}}, T) = \min (1, e^{-(f(S_{\text{next}}) - f(S_{\text{curr}})) / kT}) \) where \( f(S_i) \) decreases with the optimality (sometimes referred to as the energy) of the solution being examined and a temperature function \( T \) that decreases with each step. The probability density of this is \( \frac{e^{-f(x)/kT}}{z} \).

The goal of this process is that when \( T \) is high at the beginning of the process, solutions will be chosen more randomly; on the other hand, as the process continues and \( T \) becomes lower, the probability distribution will begin to be dominated by more optimal solutions. In theory, for a slow enough decrease of \( T \), simulated annealing will find the optimal solution every time.

Python Simulated Annealer Documentation

Creating an optimization problem and corresponding states:

You’ll need to create a python script, starting with `from SimulatedAnnealing import *`. From there, you’ll need to create the following:

- A class that implements SAProblem and the following methods:
  ```python
class MySAProblem(SAProblem):
    def __init__(self, some_other_parameters):
        # Initialize parameters for problem.
        pass

    def __str__(self):
        return "MySAProblem description"

    def generate_initial_state(self):
        # Generate an initial state, probably randomly.
        return MySAState(some_parameters)
  ```

- A class that implements SAState and the following methods:
  ```python
class MySAState(SAState):
    def __init__(self, some_other_parameters):
        # Initialize parameters and state.
        pass

    def __str__(self):
        return "MySAState description"

    def get_neighbor(self):
```
# Generate a new neighbor to this state.
return MySAState(some_new_parameters)

def representation(self):
    # Represent the state in some way. This is called occasionally by the simulated annealing loop. You can put graphics in here if you'd like them, or just print something.
    print "state with parameter",some_parameter

def energy(self):
    # Returns the energy of the state as a floating point. Lower is more optimal.
    return 0.0

• A main function that does something like this:
if __name__ == '__main__':
    # Create a simulated annealer with the problem you've created
    sa = SimulatedAnnealer(MySAProblem(some_values),
                              beta=1, # Scale the temperature function. Lower values make the algorithm more conservative.
                              stepMax=500, # Step limit.
                              energyMax=0.0, # Maximum final energy. The algorithm will stop after it crosses this threshold.
                              printfreq=20, # (OPTIONAL) Display an update every 20th step.
                              sleeptime=0.0, # (OPTIONAL) Sleep time between printouts. This lets you see what's happening if the algorithm is running to quickly.
                              temp=custom_temperature_function) # (OPTIONAL) A custom temperature function. If not specified, this will be linear. See the travelling salesman problem source code for an example.
    raw_input("Press enter to begin...")
    sa.run()
    raw_input("Press enter to finish...")

Once you fill in these methods, you can run your simulation.

Generating an instance of the simulated annealing algorithm and running it:

If you followed the instructions above, you can simply execute this command:
python ./MySAProblem.py
This will run your simulation!

Appendices:
Sample script that simulates the travelling salesman problem:
##
## Travelling Salesman for Simulated Annealing
## Nathan Germer
## 12/19/2012
##
from SimulatedAnnealing import *
import random
from random import random as rand
from math import sqrt
from Tkinter import *

class SATravellingSalesman(SAProblem):
    def __init__(self, numPoints=10, uidim=400):
        #generate points
        self.pts = []
        for i in xrange(numPoints):
            self.pts.append((rand(),rand()))

        #generate distance array
        self.dist = []
        for i in xrange(numPoints):
            self.dist.append([])
            for j in xrange(numPoints):
                pIx, pIy = self.pts[i]
                pJx, pJy = self.pts[j]
                dist = sqrt(pow(pIx-pJx,2)+pow(pIy-pJy,2))
                self.dist[i].append(dist)

        self.drawing = TSDrawKit(self.pts, uidim)

        print "Done generating travelling salesman problem."

    def __str__(self):
        return "TSP with "+str(len(self.pts))+" points."

    def generate_initial_state(self):
        random_ordering = range(0,len(self.pts))
        random.shuffle(random_ordering)
        return TSState(self.dist, random_ordering, self.drawing)

class TSState(SAState):
    def __init__(self, dists, order, drawkit):
        self.dists = dists
        self.order = order
        self.drawing = drawkit

        #compute the distance for this instance
        self.path_length = 0.0
        for i in xrange(len(order)-1):
            self.path_length += self.dists[self.order[i]][self.order[i+1]]
            self.path_length += self.dists[self.order[0]][self.order[len(order)-1]]

    def __str__(self):
        return "Path ordering: "+str(self.order)+" with length "+str(self.path_length)

    def get_neighbor(self):
        # generate index
        left = random.choice(range(0,len(self.order)))
        right = random.choice(range(0,len(self.order)))
        if left > right:
tmp = right
right = left
left = tmp

# reverse segment:
new_order = list(self.order)  # copy list (don't change the original)
new_order[left:right+1] = reversed(new_order[left:right+1])

return TSState(self.dists, new_order, self.drawing)

def representation(self):
    # draw the path on screen.
    self.drawing.draw_paths(self.order)
    self.drawing.update_energy(self.energy())
    self.drawing.update_tk()

def energy(self):
    return self.path_length

class TSDrawKit:
    def __init__(self, pts, dim=400):
        self.dim = dim

        self.master = Tk()
        self.w = Canvas(self.master, width=dim+10, height=dim+30)
        self.w.pack()

        # show points on canvas!
        self.drawing_coords = []
        for point in pts:
            px, py = point
            newpoint = (int(px*dim+5), int(py*dim)+5)
            self.drawing_coords.append(newpoint)
            npx, npy = newpoint
            self.w.create_rectangle(npx-2, npy-2, npx+2, npy+2, fill="red")

        self.text = self.w.create_text((5,dim+5), anchor=NW, text="Ready."

        self.drawn_paths = []
        self.update_tk()

    def update_energy(self, energy):
        self.w.delete(self.text)
        string = "Energy: " + str(energy)
        self.text = self.w.create_text((5,self.dim+30), anchor=SW, text=string)

    def draw_paths(self, order):
        for path in self.drawn_paths:
            self.w.delete(path)

        for i in range(len(order)-1):
            plx, ply = self.drawing_coords[order[i]]
p2x, p2y = self.drawing_coords[order[i+1]]
            line = self.w.create_line(plx,ply,p2x,p2y)
            self.drawn_paths.append(line)
#include the return trip
plx, ply = self.drawing_coords[order[0]]
p2x, p2y = self.drawing_coords[order[len(order)-1]]
line = self.w.create_line(plx,ply,p2x,p2y)
self.drawn_paths.append(line)

def update_tk(self):
    self.master.update()

def logarithmic_temperature_schedule(progress):
    from math import e, pow
    return (pow(e,1.0-progress)-1.0)/(e-1.0)

if __name__ == '__main__':
    sa = SimulatedAnnealer(SATravellingSalesman(numPoints=10, uidim=400),
                           beta=1, stepMax=500, energyMax=0.0, printfreq=1, sleeptime=0.05,
                           temp=logarithmic_temperature_schedule)
    raw_input("Press enter to begin...")
    sa.run()
    raw_input("Press enter to finish...")

Source code for Simulated Annealer and the basic interfaces:
##
## Simulated Annealing Module
## Nathan Germer
## 12/19/2012
##
## from math import e
from random import random as rand
from time import sleep

# Provides a default temperature schedule.  Can be overridden.
#   Input: progress as a floating point value: 0.0 at start, 1.0 at
#   end.
#   Output: temperature.  Should be some arbitrary value at start (in
#           this case, 1.0) and 0.0 at end.
def defaultTempSched(progress):
    return 1.0 - progress

#This is the simulated annealing algorithm.
class SimulatedAnnealer:
    def __init__(self, problem, beta, stepMax, energyMax, printfreq=20,
                 sleeptime=0.2, temp=defaultTempSched):
        self.problem = problem
        self.state = problem.generate_initial_state()
        self.energy = self.state.energy()
        self.beta = beta
        self.stepMax = stepMax
        self.energyMax = energyMax
        self.temp = temp
        self.step = 0
        self.pf = printfreq
        self.sleeptime = sleeptime
        self.done = False
        print "Done setting up simulated annealer for",self.problem
        print "The initial state is:"
self.state.representation()

# Run the simulated annealing problem.
def run(self):
    if self.done:
        print "You've already completed the simulation!"
        return

print "Running simulation. Target energy:", self.energyMax

while self.step<self.stepMax and self.energy>self.energyMax:
    self.T = self.temp(float(self.step)/float(self.stepMax)) / self.beta
    considerState = self.state.get_neighbor()
    if self.P(self.state.energy(), considerState.energy(),
               self.T)>rand():
        self.state = considerState
        self.energy = self.state.energy()

    # conditional progress report:
    if (self.step%self.pf) == 0:
        self.midwayPrintState()

    # end of loop
    self.step+=1

print '\nSimulation complete in', self.step, 'steps with result:
self.state.representation()
self.done=True
return self.state

# Helper methods:
def midwayPrintState(self):
    print '\nStep:', self.step,
    print "Current Temp:", self.T
    self.state.representation()
    sleep(self.sleeptime)

    def P(self, currStateEnergy, possibleStateEnergy, T):
        calcValue = pow(e, (currStateEnergy-possibleStateEnergy)/self.T)
        return min(1.0, calcValue)

## Interfaces.
##
## Defines a simulated annealing problem
class SAProblem:
    def __init__(self):
        print "created a default simulated annealing problem..."

    def __str__(self):
        return "DEFAULT PROBLEM"

# Generates the initial state for the problem
def generate_initial_state(self):
    return SAState()
# Defines a state for the simulated annealing problem.
# The default optimization problem is a random walk towards 0.
class SAState:
    def __init__(self, value=5):
        print "created a default simulated annealing state..."
        self.value = value

    def __str__(self):
        #return "DEFAULT STATE"
        return "value is "+str(self.value)

    # Generates a neighbor at random.
    def get_neighbor(self):
        #return self
        if rand() < .5 and self.value > 0:
            return SAState(self.value - 1)
        else:
            return SAState(self.value + 1)

    # This function is called periodically during execution. If you have
    # a visualization for your problem, you can update it here.
    def representation(self):
        print self

    # This defines the energy for your state. The optimization aims for
    # lower values.
    def energy(self):
        return float(self.value)

# This will run a default simulation.
if __name__ == '__main__':
    sa = SimulatedAnnealer(SAProblem(), beta=.2, stepMax=100,
                            energyMax=0.0)
    sa.run()