Teaching Stochastic Local Search

Todd W. Neller
Department of Computer Science
Gettysburg College
Gettysburg, PA 17325, USA
tneller@gettysburg.edu

Abstract

This paper outlines an experiential approach to teaching stochastic local search. Using the *Tale of the Drunken Topographer* as a running analogy, students are led from the implementation of a hill descent algorithm through small, motivated modifications to a simple implementation of simulated annealing. Supplementary applets allow students to experiment with temperature and gain understanding of its importance in the annealing process. Challenge problems complete this brief but rich introduction to stochastic local search.

Introduction

Our approach to teaching the core concepts of stochastic local search is largely experiential and assumes an interactive lab teaching environment. This approach may, of course, be customized to a variety of teaching environments. If it is not possible to present this material in an interactive lab setting, one might provide a lab manual which, like this paper, guides the student through independent discovery.

The following is an outline of an approach that has been successfully used in several iterations of introductory artificial intelligence courses:

- Define the stochastic local search problem.
- Introduce and explain a programming abstraction of the problem, using simple examples.
- As a class, develop an algorithm for hill descent.
- Test hill descent with the given example problems to witness local minima.
- Add to hill descent a fixed probability of accepting an uphill step and continue experimentation varying this acceptance rate.
- Copy this randomized hill descent implementation and modify it to perform simulated annealing, introducing concepts and terminology.
- Experiment with applets to see the impact of the annealing schedule.
- Offer a set of simple challenge problems, allowing students a choice of optimization projects.

Copyright © 2005, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

We elaborate on these steps in the following sections. Relevant code, problem descriptions, and a sample syllabus can be found at our website for stochastic local search teaching resources[1]. One can, of course, base an entire course on stochastic local search. A suitable text, (Hoos & Stützle 2005) has recently been published.

Defining the Problem

The goal of stochastic local search is to seek a *state* from a set of states S that optimizes some measure. We call this measure the state's *energy*, denoted $e : S \rightarrow \mathcal{R}$, and we seek a state with *minimal* energy. Our task is then to seek state s minimizing $e(s)$, that is, $\arg\min_{s \in S} e(s)$. In practice, it is often the case that we are only able to find an approximately optimal state. For each state, there is a neighborhood $N(s)$ which defines those states we may look to next after s in our search. We choose a successor state from $N(s)$ (which is in some sense "local" to s) stochastically. We call such a search *stochastic local search.*

It is both important to give a clear formal definition of the problem *and* to provide the student with an intuitive visual analogy to aid understanding. With a visual aid such as a small warped square of egg crate foam (or some other suitable bumpy surface with many local minima), our favorite approach is to tell **The Tale of the Drunken Topographer**:

> Once upon a time there was a drunken topographer. One night, after too many drinks, he became obsessed with the idea of finding the lowest point of a very hilly region, so he hired a plane and parachuted into the region. Having landed safely, he detached his parachute and began to stagger about randomly.

At this point, explain to the class that each point on the terrain is a state. Also, explain that the height of each point corresponds to the energy of the state. This is easily remembered, as the simple formula for potential energy of a mass above the Earth is mgh, where m is mass, g is the acceleration due to gravity, and h is height. Height is thus proportional to potential energy.

> Although the drunk was obsessed, he was also extremely tired. Although he staggered randomly, he was too tired to take uphill steps. If, in placing his foot

[1] http://cs.gettysburg.edu/%7Etneller/resources/sls/

down in the darkness, he felt that it was a step uphill, he set his foot elsewhere in search of a level or downhill step.

One weakness of this analogy is that the angle of one's foot would provide local slope (gradient) information to guide the next step downhill. In stochastic local search, we do not generally assume that the current state provides any indication of downward direction to another state. We can explain that the drunk cannot sense his feet, but can sense the difference of the height of his feet.

The State Interface

At this point, we are ready to introduce students to a programming abstraction of the problem. The `State` interface, given here in the Java language, is the set of all methods[2] needed for the stochastic local search algorithms we will explore.

⟨*State.java*⟩≡
```
public interface State extends Cloneable {
     void step();
     void undo();
     double energy();
     Object clone();
}
```

The `step` method changes the object's state, in effect taking a random step "nearby" on the landscape of the state space. When using the term "nearby", we refer to the *local* nature of stochastic *local* search. In defining `step`, we implicitly define what is local. That is, we impose a structure on the space for our search. If we simply pick a new random state uniformly from the entire state space, we defeat the intended locality of the search and degenerate to simple generate and test. However, if we choose slight modifications to the state which will generally not cause large changes to its energy, then we define an energy "landscape" to search. Finally, we note that the repeated application of `step` should allow the possibility to transition from any one state to any other state. We do not wish to be trapped in any part of the state space.

The `undo` method undoes the changes made by `step`. In effect, it rejects the previous step taken. For this reason, the `step` method must store sufficient information to undo its changes. The `undo` method makes use of this change information to restore the previous state. We make the restriction on our algorithms that two calls to `undo` will always have at least one intervening call to `step`. In other words, we will never seek to undo more than one previous change.

The `energy` method returns the energy of the current state. This is the measure we wish to minimize. If we have a utility of a state $U(s)$ that we wish to maximize, we can simply define energy $e(s) = -U(s)$.

The `clone` method should make a deep copy of the state which is unaffected by future changes such as `step` calls. This is used in keeping track of the minimum state during search.

After explaining the interface, we then provide two examples of State implementations. The first is a variation

[2] i.e. functions, procedures, etc.

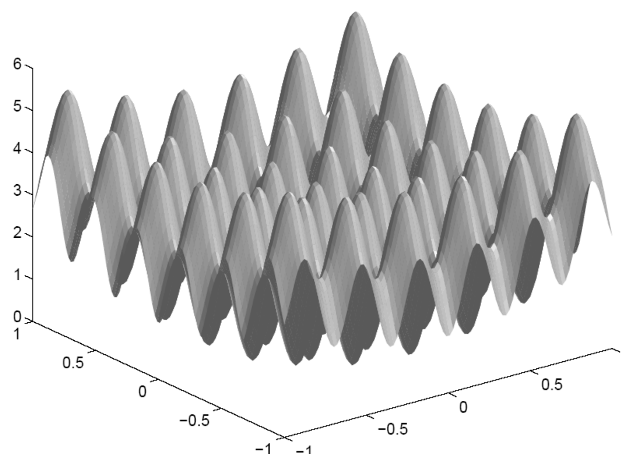


Figure 1: The Rastrigin function.

of the Rastrigin function (see Figure 1), which is a circular paraboloid with a sinusoidal surface, much like egg crate foam curved into a bowl shape.

$$\text{energy}(x, y) = x^2 + y^2 - \cos(18x) - \cos(18y) + 2$$

This function has many local minima, but the global minimum is at (0,0) with energy 0. The implementation is simple. We first define the variables of our class:

⟨*Define variables*⟩≡
```
public static final double STDDEV = .05;
public static java.util.Random random
               = new java.util.Random();
public double x;
public double y;
public double prevX;
public double prevY;
```

The static `STDDEV` parameter and `random` number generator are used in `step`. The current point on the Rastrigin function is (`x`, `y`), and the previous point is (`prevX`, `prevY`).

Two constructors allow us to start our search at any point, or $(10, 10)$ by default.

⟨*Construct initial states*⟩≡
```
public Rastrigin() {
    this(10.0, 10.0);
}

public Rastrigin(double x, double y) {
    this.x = x;
    this.y = y;
    prevX = x;
    prevY = y;
}
```

The `step` method stores the previous (x, y) position, and adds a random Gaussian value to each with a mean of 0 and a standard deviation of `STDDEV`. In this case, the neighborhood $N(s)$ is the entire state space S, and the Gaussian

probability distribution over these neighbors is what provides search locality.

⟨*Stochastically choose the next state*⟩≡
```
public void step() {
     prevX = x;
     prevY = y;
     x += STDDEV * random.nextGaussian();
     y += STDDEV * random.nextGaussian();
}
```

The method to undo the change of `step` is simple:

⟨*Undo the state change*⟩≡
```
public void undo() {
     x = prevX;
     y = prevY;
}
```

The `energy` is the Rastrigin function:

⟨*Compute energy*⟩≡
```
public double energy() {
     return x * x + y * y - Math.cos(18 * x)
             - Math.cos(18 * y) + 2;
}
```

A copy of this `Rastrigin` state is accomplished simply:

⟨*Copy state*⟩≡
```
public Object clone() {
     Rastrigin copy = new Rastrigin(x, y);
     copy.prevX = prevX;
     copy.prevY = prevY;
     return copy;
}
```

Finally, we provide a `toString` method for convenient display of the state.

⟨*Return a state description*⟩≡
```
public String toString() {
     return "(" + x + ", " + y + ")";
}
```

All together, these components define our example `State` implementation called `Rastrigin`:

⟨*Rastrigin.java*⟩≡
```
public class Rastrigin implements State {
     ⟨Define variables⟩
     ⟨Construct initial states⟩
     ⟨Stochastically choose the next state⟩
     ⟨Undo the state change⟩
     ⟨Compute energy⟩
     ⟨Copy state⟩
     ⟨Return a state description⟩
}
```

The second example `State` implementation we provide is a bin packing problem state. This implementation is not described here, but is available at our website for stochastic local search teaching resources[3].

[3] http://cs.gettysburg.edu/%7Etneller/resources/sls/

Hill Descent

As a class, one can now interactively develop a simple algorithm for hill descent. We describe one such implementation here. First we define variables for the current search state (`state`), the current minimum energy search state (`minState`), and the associated energy for each (`energy` and `minEnergy`).

⟨*Define* `HillDescender` *variables*⟩≡
```
private State state;
private double energy;
private State minState;
private double minEnergy;
```

We then initialize these variables in the constuctor which is given an initial state.

⟨*Construct with initial state*⟩≡
```
public HillDescender(State initState) {
     state = initState;
     energy = initState.energy();
     minState = (State) state.clone();
     minEnergy = energy;
}
```

The search method is given `iterations`, a number of steps to take, after which it returns the minimum energy state (`minState`) from all iterations.

⟨*Search for minimum state*⟩≡
```
public State search(int iterations)
{
     for (int i = 0; i < iterations; i++) {
          if (i % 100000 == 0)
               System.out.println(minEnergy
                              + "\t" + energy);
          state.step();
          double nextEnergy = state.energy();
          if (nextEnergy <= energy) {
               energy = nextEnergy;
               if (nextEnergy < minEnergy) {
                    minState
                      = (State) state.clone();
                    minEnergy = nextEnergy;
               }
          }
          else
               state.undo();
     }
     return minState;
}
```

To help students get a feel for the progress of the search, we sample the minimum and current energies every 100,000 iterations and print them in two columns. With each iteration, we change the state with `state.step()`. If the state has lesser or equal energy than before, we update our `energy` to that value and check if this state is possibly the best (minimal energy) state seen so far. If, however, the state has greater energy than before, we `undo` the change. Finally, we return the minimal state. Together, these components define a `HillDescender` class.

⟨*HillDescender.java*⟩≡
```
public class HillDescender {
```

⟨*Define* HillDescender *variables*⟩
⟨*Construct with initial state*⟩
⟨*Search for minimum state*⟩
}

Local Minima

Having implemented `HillDescender` students should write test programs to see how well this approach performs with the example problems. In particular, have students note the behavior with the Rastrigin function, and have them seek to explain why a close approximation to the global minimum is not often found.

It is best to have the students arrive at an experiential understanding of *local minima* by observing the search getting trapped in them. Introduce the term *local minima* and have students propose means of overcoming them.

Hill Descent with Random Uphill Steps

> The dead-tired drunk, making only downhill steps, will quickly find himself stuck in a ditch, but this ditch will not necessarily be at the lowest height. Let us suppose that the drunk is not so extremely tired, and will take a step uphill with some probability.

We now have the students make a trivial modification to their code to allow some uphill steps. Whereas before, our condition for accepting a next state was `(nextEnergy <= energy)`, we now use the new condition:

```
(nextEnergy <= energy
 || random.nextDouble() < acceptRate)
```

where `random` is a `java.util.Random` object and `nextDouble` returns a `double` in the range $[0, 1)$. That is, we always accept states with lesser or equal energy, and we accept higher energy states with some given probability `acceptRate`. Note that when `acceptRate` is set to 0, we have the same strict hill-descending behavior as before.

> Taking this to an extreme, let us imagine the *super drunk* who has unlimited stamina and will always take a step uphill. This drunk will freely wander the landscape, neither preferring to go up or down.

Have students set `acceptRate` to 1, and observe the poor quality of the resulting searches for the example problems. This extreme is merely a random walk. Have students experiment with different values for `acceptRate`, e.g. $.1, .01, .001, .0001$, etc. for each example problem until an improvement in average performance over the extremes is observed. This experimentation can go beyond the lab setting. One might assign students to vary `acceptRate` with all other parameters fixed, performing many runs at each setting and plotting the median value.

Simulated Annealing

> We see that a drunk with at least some energy to climb out of local minima is more likely to find lower local minima. However, our drunk is not at all particular as to whether the step uphill is big or small. We now consider a drunk who is more likely to take a small uphill step than a large uphill step.

In the case of an uphill step, our simple condition was `random.nextDouble() < acceptRate`. Now we wish to make this condition dependent upon how far uphill the step will take us. Natural systems which seek a minimal energy state, such as metals cooling or liquids freezing, were modeled by Metropolis et al (Metropolis *et al.* 1953) in what has come to be known as the Metropolis algorithm. In their simulation, an uphill step with energy change ΔE is accepted with probability $e^{(\frac{-\Delta E}{kT})}$, where k is Boltzmann's constant (1.38×10^{-23} joules/kelvin), and T is the *temperature of the system*. In practice, we can and will ignore Boltzmann's constant, compensating with our temperature parameter. That is, we can choose a much smaller temperature parameter as if we are multiplying it by k. This saves us unnecessary floating point operations in computing the acceptance condition. Thus our new acceptance condition is:

```
(nextEnergy <= energy
 || random.nextDouble()
    < Math.exp((energy - nextEnergy)
               / temperature))
```

Students should discuss what happens to this acceptance probability for extremes of ΔE and T. This randomized hill descent code should then be copied to a class `SimulatedAnnealer`. Replace the parameter `acceptRate` with `temperature`. Students can experiment with different `temperature` settings just as they experimented with different `acceptRate` settings. We next have students learn about the importance of temperature empirically through the use of applets.

Traveling Salesman Problem The first applet[4] is shown in Figure 2. Each instance of the traveling salesman problem (TSP) contains 400 cities randomly placed on a 350 by 350 grid. A tour is represented as a permutation of the list of cities, and the energy function $e(s)$ is simply the total distance traveled along the tour. We use the next state generation function of Lin and Kernighan described in (Kirkpatrick, Gelatt, & Vecchi 1983) in which a section of the current tour is reversed at each step.

The "Anneal" checkbox toggles the search on/off. Begin by setting the temperature to the minimum, sliding the slider bar to the extreme left. Check "Anneal" to begin the annealing. Have students observe and write down the energy value ("Length") of this local minimum. Explain that when the temperature is very low, the acceptance rate approaches 0 and we have our "dead-tired drunk's" hill-descending behavior. Slide the temperature bar to the extreme right. This corresponds to the "super drunk's" random walk. Now have students seek a better solution than the previous local minimum by slowly moving the slider bar to the left. Let them experiment for a good while. This experience can find no substitute in books, on the board, or from the best of anecdotes. Often students will come to understand concepts such as *simulated tempering* or *restarts* on their own.

After this experimentation, check the "Clusters" box and press the new problem button. This is the same clustered

[4] http://cs.gettysburg.edu/%7Etneller/resources/sls/tsp/

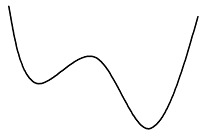
TSP problem as presented in (Kirkpatrick, Gelatt, & Vecchi 1983) where the 400 cities are grouped into nine clusters. Have students note that the paths *between* clusters are best minimized in a different, higher temperature range than that where the paths *within* clusters are best minimized. Gross features of the problem are optimized at higher temperatures than the details of the problem. The need for different temperatures is further reinforced by an applet demonstrating a course scheduling problem.

Course Scheduling Problem The second applet[5] is shown in Figure 3. Each randomly generated instance of our simple course scheduling problem consists of 2500 students, 500 courses, and 12 time slots. Each student is enrolled in 5 unique random courses. Each state of the problem is represented as a list containing the time slot for each course. Next states are generated by randomly changing the time slot of one course. The energy function $e(s)$ is defined as the total number of course conflicts, i.e. the total number of class pairs in each student's enrollment which occur in the same time slot.

For this demo, the vertical axis of the pane represents conflicts. As the pane scrolls horizontally with time, and two lines evolve with the optimization. The blue curve is a evolving plot sampling the current state energy. The lower red curve represents the best (minimum) state energy over time. By experimenting with this applet, students will see that the best progress is made by adjusting the temperature such that the distribution of current state energies is not too far from the current best state. They will find it advantageous to adjust the temperature over time to minimize scheduling conflicts.

Annealing Schedules

Next display a simple curve with two local minima of different heights:



> Beginning his quest with great vigor, the drunk will freely wander among basins of local minima. When he becomes sufficiently tired, he will wander to the bottom of one such basin. If he becomes tired very gradually, there will come a time when he is more likely to climb up out of a high basin into a low one than vice versa. How low a basin the drunk finds depends on how long the drunk wanders and how gradually he tires.

Now that we have motivated *simulated annealing* (Kirkpatrick, Gelatt, & Vecchi 1983), we make one final simple modification to the student's code. They will have already fixed a temperature parameter to an initial value. All they need to do to have a simple geometric annealing schedule (a.k.a. cooling schedule) is to introduce a `decayRate` parameter, set to be slightly less than 1, and add the following line to the end of the search iteration:

[5] http://cs.gettysburg.edu/%7Etneller/resources/sls/scheduling/

```
temperature = temperature*decayRate;
```

The students have now implemented a simple simulated annealing algorithm with a geometric annealing schedule. They will need some time to experiment and discover how close `decayRate` should be to 1 to cool at a reasonable rate. Have them try `decayRate` values such as .9, .99, .999, etc.

Challenge Problems

The next phase of the student's learning is to take on challenge problems with these algorithms. Before, the student was given two implementations of the `State` interface. Now it is time for the student to gain exercise in the implementation of the `State` interface. We recommend presenting a collection of challenge problems and allowing students to choose a few as homework. These are some of the problems we have assigned.

- **The Traveling Salesmen Problem** - This can optionally be modified as in (Press *et al.* 1992) to include a north-south river in the center which has an associated reward or penalty for each crossing. This can also be modified as in (Kirkpatrick, Gelatt, & Vecchi 1983) to include clustering of cities.
- **Course Scheduling Problem** - (described above)
- **N-Queens Problem** - Place n queens on an n-by-n chessboard such that no two queens share the same row, column, or diagonal.
- **Stinky Queens Problem** - The queens now repel one another. Place n queens on an n-by-n chessboard such that the minimum Euclidean distance between two queens is maximized.
- **Tree Planting Problem** - (a continuous variation of the Stinky Queens problem) Plant n trees within a circular plot such that all trees can grow to the same maximal radius before one tree's canopy touches another or crosses the plot boundary. In other words, how can one place n non-overlapping circles with radius r within a unit circle such that r is maximized?

There are, of course, many suitable combinatorial optimization problems that would serve as good small challenge problems. Students implement appropriate `State` classes, find good search parameters, and display their results. One can optionally have students share their problem solving experiences and results in class.

Final Thoughts

There is no substitute for experience. There is a common tension in introductory AI courses between breadth and depth. Since it is most often the only exposure that a student has to AI, many instructors choose breadth over depth in order to provide students with a significant index of AI problem solving techniques.

There is the risk that such an introductory approach to a topic is like a superficial passing introduction to a person at a party. The person passes, a moment passes, and the name is forgotten. However, a meaningful discussion following an

introduction will make a person more memorable. We aim for such an introductory discussion with our course material.

Fortunately, stochastic local search is one topic where it is possible to give students a depth of understanding through experience without a major time investment that would dominate an introductory AI course. Have your students experiment with stochastic local search (e.g. using our simulated annealing applet for the traveling salesman problem). They will gain a good grasp of core concepts of stochastic local search which cannot be as effectively learned by the written or spoken word.

> "One must learn by doing the thing; for though you think you know it, you have no certainty, until you try."
> - Sophocles

This is but one set of experiences. Simulated annealing is but one stochastic local search algorithm. There are many others. If there is another stochastic local search algorithm that the reader believes is more beneficial, we would recommend evolving an implementation from a trivial algorithm (e.g. hill descent) to that algorithm, motivating each step of the evolution experientially as we have here.

The approach we present only touches lightly on a vast subject. This is in harmony with our goal. We prefer to give students a brief, positive, and richly experiential taste of the vast possibilities of stochastic local search. An entire course or independent study could focus on an introduction to stochastic local search. Indeed, there is now a suitable textbook (Hoos & Stützle 2005).

We hope that you and your students benefit greatly from our experiences and come away with an excitement for this set of power tools for the mind.

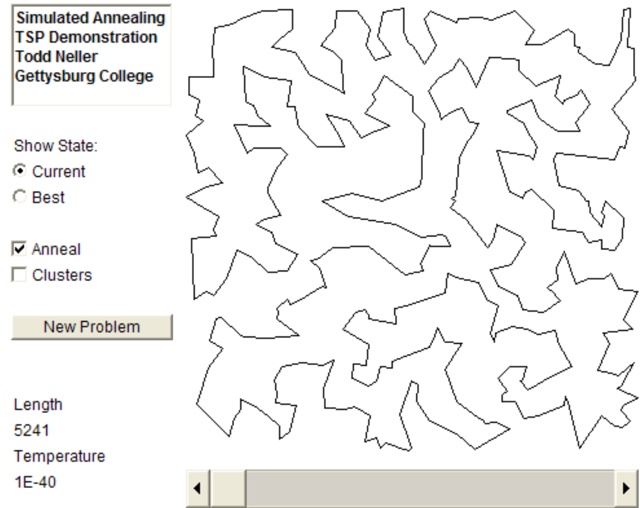


Figure 2: Traveling Salesman Applet

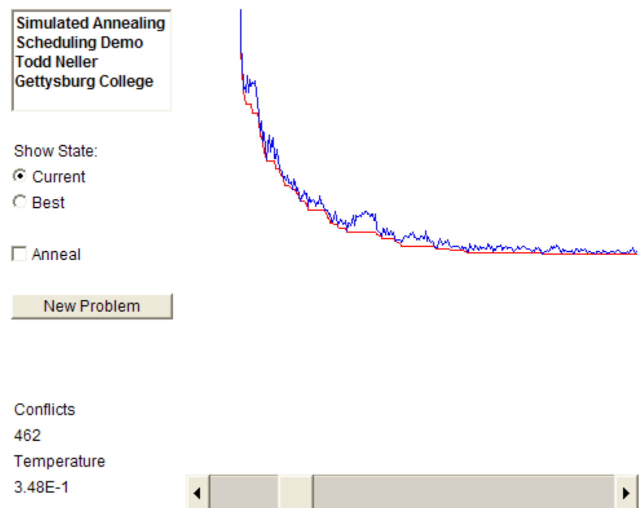


Figure 3: Scheduling Problem Applet

References

Hoos, H. H., and Stützle, T. 2005. *Stochastic Local Search: foundations and applications*. San Francisco: Morgan Kaufmann.

Kirkpatrick, S.; Gelatt, C.; and Vecchi, M. 1983. Optimization by simulated annealing. *Science* 220:671–680.

Metropolis, N.; Rosenbluth, A.; Rosenbluth, M.; Teller, A.; and Teller, E. 1953. Equation of state calculations by fast computing machines. *J. Chem. Phys.* 21(6):1087–1092.

Press, W. H.; Teukolsky, S. A.; Vetterling, W. T.; and Flannery, B. P. 1992. *Numerical Recipes in C: the art of scientific computing - 2nd Ed.* Cambridge: Cambridge University Press.