Steepest-Ascent Hill-Climbing

OCTOBER 15, 2018

Search algorithms have a tendency to be complicated. Genetic algorithms have a lot of theory behind them. Adversarial algorithms[1] have to account for two, conflicting agents. Informed search relies heavily on heuristics. Well, there is one algorithm that is quite easy to grasp right off the bat.

Imagine you are at the bottom of the hill; you have no idea where to go. A decent place to start would be to go up the hill to survey the landscape. Then, restart to find a higher a peak until you find the highest peak, right? Well, that is the entire algorithm.

Let's dig a bit deeper.

An Introduction

What is Steepest-Ascent Hill-Climbing, formally? It's nothing more than an agent searching a search space, trying to find a local optimum. It does so by starting out at a random Node, and trying to go uphill at all times.

The pseudocode is rather simple:

```
current ← Generate-Initial-Node()
```

```
while true
    neighbors ← Generate-All-Neighbors(current)
    successor ← Highest-Valued-Node(neighbors)

    if Value-At-Node(successor) <= Value-At-Node(current):
        return current

    current ← successor
```

What is this `Value-At-Node` and f-value mentioned above? It's nothing more than a heuristic value that used as some measure of quality to a given node. Some examples of these are:

- *Function Maximization*: Use the value at the function $f(x, y, \ldots, z)$.
- *Function Minimization*: Same as before, but the reciprocal: $1/f(x, y, \ldots, z)$.
- *Path-Finding*: Use the reciprocal of the Manhattan distance.
- *Puzzle-Solving*: Use some heuristic to determine how well/close the puzzle is solved.

The best part? If the problem instance can have a heuristic value associated with it, and be able to generate points within the search space, the problem is a candidate for Steepest-Ascent Hill-Climbing.

Implementing Steepest-Ascent Hill-Climbing

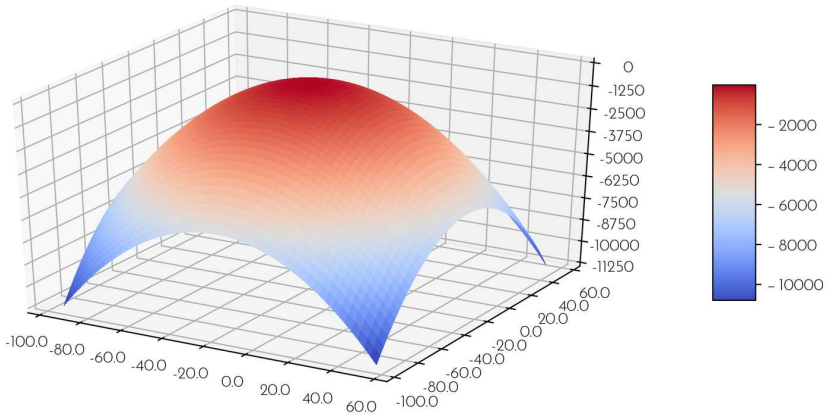
For this problem, we are going to solve an intuitive problem: function maximization. Given a function $z = f(x, y)$, for what values of x, y will z be the largest? To start, we are going to use a trivial function to maximize:

$$z = -x^2 - y^2$$

We see it is nothing more than a paraboloid. Furthermore, since it is infinite, we are going to restrict the domain to

$x, y \in \mathbb{Z}^+ : -100 \leq x, y \leq 100$; therefore, we only have integer values between $(-100, 100)$.

1. Algorithms used in games, where a player searches for an optimal move against an opponent. ↩



$$z = -x^2 - y^2$$

So, let's begin.

The Representation

Because we will be searching throughout a search space, we will need some representation of a state. For our particular problem instance, it's very easy: the points (x, y). Also, we will need to represent the f value, so we create an auxiliary class as well.

```
class Node:
    """A node in a search space (similar to a point (x, y)."""

    def __init__(self, x, y):
        self.x = x
        self.y = y
```

```
class Function:
    """A function and its respective bounds."""

    def __init__(self, function, x_bounds, y_bounds):
        ...
```

```
    def __call__(self, node):
        ...

    @property
    def x_bounds(self):
        """Get the x bounds of the function.

        Returns:
            tuple<int, int>: The x bounds of function in the fo
        """
        ...

    @property
    def y_bounds(self):
        """Get the y bounds of the function.

        Returns:
            tuple<int, int>: The y bounds of function in the fo
        """
        ...
```

That will be all that we need for our purposes.

Steepest-Ascent Hill-Climbing

As we saw before, there are only four moving pieces that our hill-climbing algorithm has: a way of determining the value at a node, an initial node generator, a neighbor generator, and a way of determining the highest valued neighbor.

Starting with the way of determining the value at a node, it's very intuitive: calculate the value $z = f(x, y)$.

```
class HillClimber:
    """A steepest-ascent hill-climbing algorithm."""

    def __init__(self, function):
        self.function = function

    def _value_at_node(self, node):
        return self.function(node)
```

The initial node can simply be taken as a random (x, y) in their respective bounds.

```
    def _initial_node(self):
        x = randint(self.function.x_bounds[0], self.function.x_
        y = randint(self.function.y_bounds[0], self.function.y_

        return Node(x, y)
```

Generating neighbors is actually quite simple as well: because our domain is limited to integers, we can simply look at the four cardinal directions (and make sure we won't be breaking the bounds when we do). Also, we randomize the neighbors, to make things more interesting[1].

```
    def _generate_all_neighbors(self, node):
        x, y = node.x, node.y

        nodes = [Node(x, y)]

        if x < self.function.x_bounds[1]:
            nodes.append(Node(x + 1, y))
        if x > self.function.x_bounds[0]:
            nodes.append(Node(x - 1, y))
        if y < self.function.y_bounds[1]:
            nodes.append(Node(x, y + 1))
        if y > self.function.x_bounds[0]:
            nodes.append(Node(x, y - 1))

        shuffle(nodes)
        return nodes
```

Finally, to get the highest value node, it's fairly straightforward:

```
    def _highest_valued_node(self, neighbors):
        max_point = neighbors[0]

        for point in neighbors[1:]:
            if self._value_at_node(point) > self._value_at_node
                max_point = point

        return max_point
```

Piecing all this together, we get our Steepest-Ascent Hill-Climber:

```
  def climb(self):
        current_node = self._initial_node()

        while True:
            print("Exploring Node({}, {})".format(current_node.
            neighbors = self._generate_all_neighbors(current_no
            successor = self._highest_valued_node(neighbors)

            if self._value_at_node(successor) <= self._value_at
                return current_node

            current_node = successor
```

Does it work? Exactly as planned.

```
Exploring Node(5, -88)
...
Exploring Node(5, -67)
...
Exploring Node(5, -47)
...
Exploring Node(5, -27)
...
Exploring Node(5, -4)
Exploring Node(4, -4)
Exploring Node(3, -4)
Exploring Node(3, -3)
Exploring Node(2, -3)
Exploring Node(2, -2)
Exploring Node(1, -2)
Exploring Node(1, -1)
Exploring Node(1, 0)
Exploring Node(0, 0)
```

However, this was too easy. We had a function with one local optimum. Let's make things interesting.

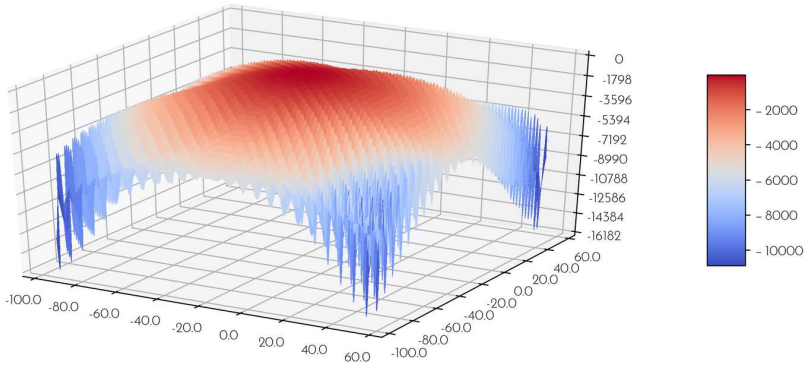
Optimizing Steepest-Ascent Hill-Climbing

Suppose we keep our previous domain, but we change our function to the following:

$$z = -(x^2 + y^2) + x\,y\,\cos x\,\sin y$$

This function isn't quite as intuitive to visualize, please reference the figure. Essentially, it's what we had before, but *thousands* of local optimum when we get further from the center. Our previous Hill-Climbing would absolutely get destroyed by that function.

1. If the neighbors are always generated deterministically, there might occur a sequence of ties when generating the highest-valued node. We randomize the neighbors so a random piece will be chosen in the tie-breaker. ↩



$$z = -(x^2 + y^2) + x\,y\,\cos x\,\sin y$$

To alleviate this, we are going to use two optimizations:

1. Instead of taking the steepest uphill move, we are going to simply take a random, uphill move (known as Stochastic Hill-Climbing).
2. When we get stuck, we are going to restart the search (known as Hill-Climbing With Restarts).

Stochastic Hill-Climbing

Updating the algorithm is fairly simply, all the previous mechanics are inheritable, just swap out `_highest_valued_node` with a stochastic version.

```
class StochasticHillClimber(HillClimber):
    """A stochastic steepest-ascent hill-climbing algorithm."""

    def _get_random_uphill_move(self, current_node, neighbors):
        uphill_nodes = []

        for point in neighbors:
            if self._value_at_node(point) > self._value_at_node
                uphill_nodes.append(point)

        return current_node if len(uphill_nodes) == 0 else choi

    def climb(self):
        current_node = self._initial_node()

        while True:
            print("Exploring Node({}, {})".format(current_node.

            neighbors = self._generate_all_neighbors(current_no
            successor = self._get_random_uphill_move(current_no

            if self._value_at_node(successor) <= self._value_at
                return current_node

            current_node = successor
```

Running this algorithm, we get better results; but we can do better.

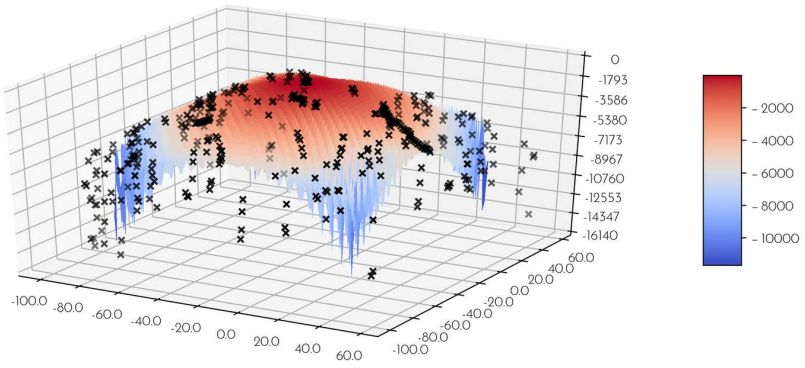
Stochastic Hill-Climbing With Restarts

For this, we simply have to restructure the `climb` function to handle generational effects (like keeping the max valued node throughout generations). Not too difficult.

```
class StochasticHillClimberWithRestarts(StochasticHillClimber):
    """A stochastic steepest-ascent hill-climbing algorithm wit

    def climb(self, number_of_generations):
        max_node = self._initial_node()

        for generations in range(number_of_generations):
```

```
            current_node = self._initial_node()

            while True:
                print("Generation {}, Exploring Node({}, {}), C
                neighbors = self._generate_all_neighbors(curren
                successor = self._get_random_uphill_move(curren

                if self._value_at_node(max_node) < self._value_
                    max_node = current_node

                if self._value_at_node(successor) <= self._valu
                    break

                current_node = successor

        return max_node
```

How did this one fare? Quite better than all the rest. Let's take a look at what the exploration process looked like.



The exploration process of Stochastic Steepest-Ascent Hill-Climbing With Restarts.

Marvelous, some got to the top, many got caught in local optimum. A global-optimum was found. A success.

Source Code

The source code be found here.

TAGGED LOCAL SEARCH AI, SEARCH-BASED AI

FreneticArray.com @IllyaStarikov @IllyaStarikov