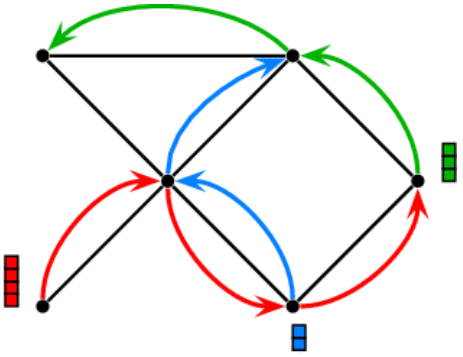
Probabilistic Approaches to Combinatorial Optimization

Ryan Shrott Follow
Aug 5, 2017 · 12 min read



Source

Introduction

Combinatorial optimization algorithms are designed to find an optimal object from a finite set of objects. In this article, I will examine two probabilistic techniques to solve such a problem: Genetic Algorithms (GA) and Simulated Annealing (SA). To demonstrate these two approaches, I will examine the raison d'être of combinatorial problems, namely, the Travelling Salesman Problem (TSP). The TSP asks the following question: "Suppose you are given a list of cities and the distances between all these cities, what is the shortest path that visits each city exactly once and returns to the starting city?". The TSP is considered to be NP-hard, which means that it can't be solved in polynomial time. That said, we can get a "close enough" solution using randomized improvement with clever heuristics.

Exact Solution to the TSP

A naive solution to the TSP would be to consider all possible paths in the state space. This solution has a run time on the order of the total number of permutations of the path. If we were trying to find an optimal path through all the state capital cities in the Unites States, we would need to examine

50! or 30 vigintilion paths, which is 304 sexdillion times the number of stars in the universe. We can reduce the runtime to exponential time by using a clever dynamic programming approach (Held-Karp Algorithm) which makes use of the fact that every subpath of a path of minimum distance is itself of minimum distance. The runtime gets reduced from 30 vigintilion to 2 quintillion for the 50 cities problem.

Genetic Algorithm

If you are unfamiliar with genetic algorithms, I highly suggest you check out my last blog which covers the basic theory with an example in game playing. A genetic algorithm can be used to solve the travelling salesman problem by evolving a population of randomly chosen paths. We can encode a representation of the problem domain in the TravelingSalesmanProblem class below. We will represent each city as a tuple containing the city name and its position specified as a location (x,y) on a grid.

```
class TravelingSalesmanProblem:

def __init__(self, cities):
        self.path = copy.deepcopy(cities)

    def copy(self):
        """Return a copy of the current board state."""
        new_tsp = TravelingSalesmanProblem(self.path)
        return new_tsp

    @property
    def names(self):
        names, _ = zip(*self.path)
        return names

    @property
    def coords(self):
        _, coords = zip(*self.path)
        return coords
```

All the code in the article can be found be found on my GitHub here.

We now need to define methods for mutation, fitness and reproduction. For mutation, we can simply swap two cities in the path randomly. The fitness is also easy to define: just use the total distance of the path (one small caveat is that you should use -1 * distance or 1/distance since shorter paths should have a higher fitness!). You also need to decide on which norm you want to use to calculate distance. For simplicity, I will use the euclidean norm.

The trickiest part is deciding on a reproduction method. We need to somehow encode the genes from both parents and create a valid path. A simple approach would be to randomly take a sub-path from the father and then fill in the remaining empty cities by linearly scanning the mother until the child path is complete [2]. The methods for applying these techniques to our TSP class are given below.

This is the fitness method using -1 times the Euclidean distance of the path:

```
def fitness(self, metric='euclid'):
        # if the length is shorter, the fitness should be higher
        # For example, if length = 10000, we return -10000
        # For example, if length = 10, we retun -10
        # Since -10 > -10000, the fitness is higher for the better
path
        def euclid(x, y):
            return ((x[0]-y[0])**2 + (x[1]-y[1])**2)**.5

        def manhattan(x,y):
            return (abs(x[0]-y[0]) + abs(x[1]-y[1]))

        def inf(x,y):
            return (max(abs(x[0]-y[0]), (abs(x[1]-y[1]))))


        if metric == 'euclid':
            norm = euclid
        elif metric == 'manhattan':
            norm = manhattan
        elif metric == 'inf':
            norm = inf

        length = 0
        coords = self.coords
        for i in range(len(coords)-1):
            length += norm(coords[i], coords[i+1])

        length += norm(coords[0], coords[-1])
        return -length
```

This is the mutation method swapping two random cities on the path:

```
def mutate(self): # in place mutation
        ind = random.sample(
                [i for i,_ in enumerate(self.path)], 2)
        # swap the cities on the path
```

```
        self.path[ind[0]], self.path[ind[1]] = self.path[ind[1]],
self.path[ind[0]]
```

Finally, the reproduction method takes an instance of the TSP class as a parameter. We apply the aforementioned breeding process and return the child TSP object.

```
def reproduce(self, partner):
        # breeds with parents being the current instance
        # and partner
        if len(self.path) != len(partner.path):
            print('Cannot breed!')
            return False
        if random.uniform(0,1) > 0.5:
            ind = sorted(random.sample(
                    [i for i,_ in enumerate(self.path)], 2))
            child_path = self.path[ind[0]:ind[1]]
            partners_added = 0
            for x in partner.path:
                if len(child_path) == len(self.path):
                    break
                if x not in child_path:
                    partners_added += 1
                    if partners_added < ind[0]:
                        child_path.insert(0, x)
                    else:
                        child_path.append(x)
        else:
            ind = sorted(random.sample(
                    [i for i,_ in enumerate(partner.path)], 2))
            child_path = partner.path[ind[0]:ind[1]]
            partners_added = 0
            for x in self.path:
                if len(child_path) == len(self.path):
                    break
                if x not in child_path:
                    partners_added += 1
                    if partners_added < ind[0]:
                        child_path.insert(0, x)
                    else:
                        child_path.append(x)
        if len(child_path) != len(set([x[0] for x in
child_path])):
            print('Invalid breeding method!')
            return False

        return TravelingSalesmanProblem(child_path)
```

We also create a shuffle method which will be used to randomly generate the initial population. It simply returns a shuffled path of the cities.

```
def shuffle(self):
        new_problem = self.copy()
        random.shuffle(new_problem.path)
        return new_problem
```

At this point, the hard work is done. We just need to apply the reproduction and mutation methods repeatedly until we get a population of very fit individuals. We can easily create a class of TSP objects to represent our population and add an evolution method which returns an evolved population. The evolution method is standard and simply combines all the work we have done so far. We also let the user specify the percentage of parents which are retained in the next evolution cycle, the percentage of lesser performing parents who are not fit (i.e. they get lucky) and the mutation rate.

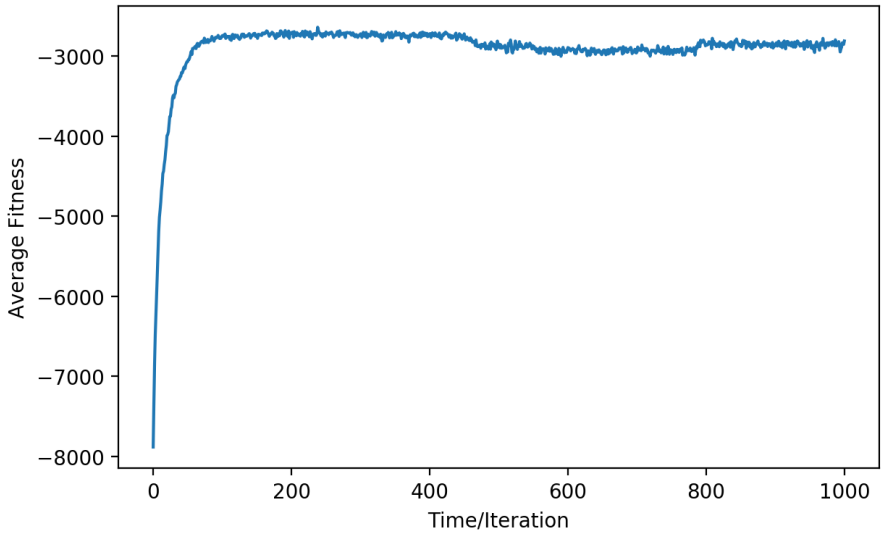
The TSP population class is written as follows:

```
class SalesmanPopulation:

    def __init__(self, population):
        self.pop = population

    def averageFitness(self):
        return np.mean([x.fitness() for x in self.pop])

    def evolve(self, retain=0.2, random_select=0.05, mutate=0.01):
        agent_performance = [(x.fitness(), x) for x in self.pop]
        sorted_perf = [x[1] for x in sorted(agent_performance,
                       key=lambda x: x[0])][::-1]
        retain_length = int(len(sorted_perf)*retain)
        parents = sorted_perf[:retain_length]

        # randomly add other agents to promote genetic diversity
        for individual in sorted_perf[retain_length:]:
            if random_select > random.random():
                parents.append(individual)

        # randomly mutate some individuals
        for i, individual in enumerate(parents):
            if mutate > random.random():
                parents[i].mutate()

        parents_length = len(parents)
        desired_length = len(self.pop) - parents_length
        children = []

        while len(children) < desired_length:
            male = random.randint(0, parents_length-1)
            female = random.randint(0, parents_length-1)
            if male != female:
                male = parents[male]
                female = parents[female]
                child = male.reproduce(female)
                children.append(child)
        evolved_population = SalesmanPopulation(parents +
children)
        return evolved_population

    def mostFitIndividual(self):
        # returns the fittest individual in the population
        fitness_dict = {x : x.fitness() for x in self.pop}
        return max(fitness_dict, key=fitness_dict.get)
```

Let's now bring everything together and actually solve the problem. We will use 30 capital cities, an initial population size of 500 and 1000 evolution cycles.

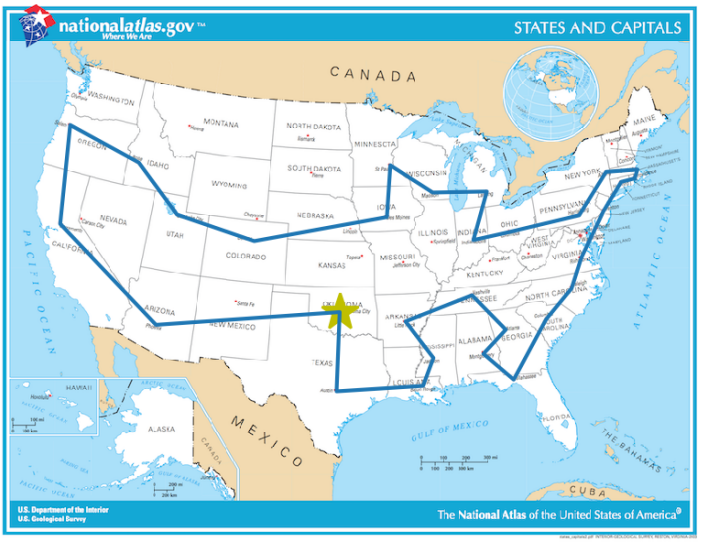
```
if __name__ == "__main__":
    num_cities = 30
    population_size = 500
    evolution_cycles = 1000
    starting_city = capitals_list[0]
    cities = capitals_list[:num_cities]
    tsp = TravelingSalesmanProblem(cities)
    show_path(tsp.coords, starting_city, w=4, h=3)
    population = SalesmanPopulation([tsp.shuffle() for _ in
range(population_size)])
    print(population.averageFitness())
    fitness_history = []
    fitness_history.append(population.averageFitness())
    for i in range(evolution_cycles):
        population = population.evolve()
        avgFitness = population.averageFitness()
        fitness_history.append(avgFitness)
        print(i, population.averageFitness())
    plt.plot(fitness_history)
    fittest_path = population.mostFitIndividual()
    print('Fittest individual has fitness
{:.2f}'.format(fittest_path.fitness()))
```

Let's plot the average fitness through time:



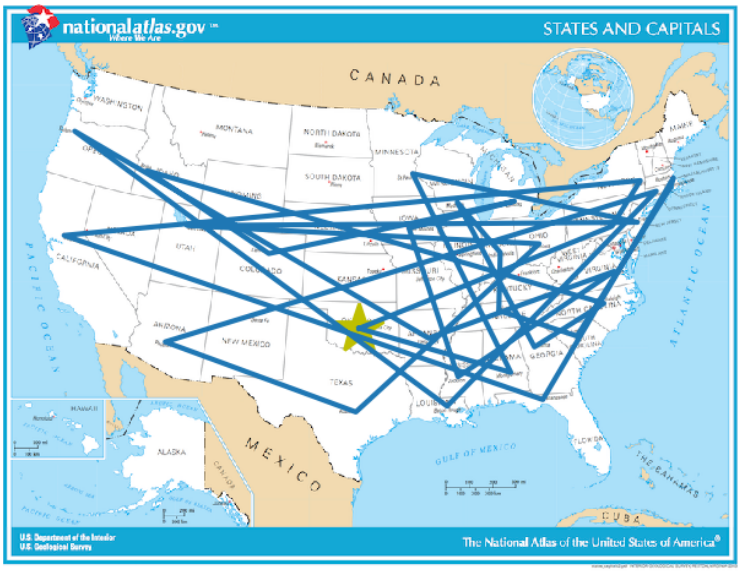
Average fitness through time

Notice that the average fitness increases very quickly at the beginning and then drops off with no sign of improvement after about 100 iterations. The fittest individual has a fitness of -2163.48 after the 1000th iteration and it gives the following route:



Genetic algorithm route generation for 30 capitals. The fitness of this path is -2163.28. The star is the starting city.

The randomized initial path length had a fitness of -8119.82 with the following route:



Before the genetic algorithm is run on 30 cities. The fitness if -8119.82. The star is the starting city.

The genetic algorithm has given us a very good solution in polynomial time. Using 200 iterations, the program computes a solution in just a few

seconds. Let's now turn our attention to another method for solving the TSP and we'll come back to genetic algorithms at the end.

Simulated Annealing

A simple hill climbing algorithm often perform poorly on combinatorial optimization problems because of its inability to get out of a local maxima. Simulated annealing attempts to combine hill climbing with random jumps, allowing the algorithm to get unstuck from local maxima. The algorithm was inspired through annealing in metallurgy: heating a sword to a very high temperature and then gradually cooling the sword, thus allowing the sword to reach a low energy crystalline state.

Annealing process in Metallurgy

Suppose we are trying to roll a ball into the lowest hill in a valley by shaking the world beneath it. If we didn't shake at all, the ball would probably roll into a local minima. If we shook too much, the ball would keep moving around and never find a minima. The trick is to shake the world just hard enough to get it out of any local minima, but not out of the global minima. The simulated annealing process starts by shaking the world hard (i.e. at a high temperature) and then gradually reducing the shaking through time (lowering the temperature gradually).

The simulated annealing algorithm differs to standard hill climbing in that it picks moves at random. If the move improves the situation then it is accepted. If it does not improve the situation, it it is accepted with probability exp(delta_E/T). Notice that if the temperature is high, then we tend to accept the poorer move more often and therefore tend to jump

more at the beginning of the algorithm. Also, if delta_E is very negative, the probability of accepting the move is close to zero. So we tend not to accept very poor moves. If the algorithm lowers the temperature slowly enough, then the algorithm outputs the global maxima with probability near 1.

We can implement the simulated annealing algorithm as follows. The function takes in an instance of the TSP problem and a temperature scheduling object (explained below).

```
def simulated_annealing(problem, schedule):
    current = problem
    t=1
    while True:
        T = schedule.expDecay(t)
        #print(T)
        if T < 1e-10:
            return current
        next_state = current.successor()
        delta_E = next_state.fitness() - current.fitness()
        if delta_E > 0:
            current = next_state
        else:
            prob = np.exp(delta_E/T)
            u = random.uniform(0,1)
            if u < prob:
                current = next_state
        t += 1
```

We now need to define how slowly the temperature should decrease with a scheduling object. It has been shown through empirical research here that simulated annealing works best when the temperature cools very quickly at the beginning and then slowly cools afterward. One popular approach is to use an exponentially decaying scheduling function of the form $T\_k = T0*alpha\^k$, where T0 is the initial temperature and alpha < 1. We can encapsulate the features of the scheduling function in the class:

```
class Schedule:
    def __init__(self, alpha, temperature):
        self.alpha = alpha
        self.temperature = temperature
    def expDecay(self, time):
        return self.alpha**(time) * self.temperature
```

We also need to define how to create random successors. I tried three approaches:

1. Random permutation of the path (Naive)
2. Random swap of two adjacent cities in the path

3. Reverse the ordering of a random subset of the path

Approach 3 was by far the most affective method and allowed the algorithm to converge to a solution much faster than methods 1 and 2. To draw an analogy to genetic algorithms, we are essentially creating a child from a single parent, which may act as the next generation. But here the population only contains a single agent. My intuition is that a random permutation is too random and won't encode the genetic material which allowed the current agent to be considered in the current cycle of the algorithm. I have a hard time explaining why method 3 outperforms method 2. Please let me know in the comments if you have any human understandable intuition for this. The successor method for the TSP class is as follows.

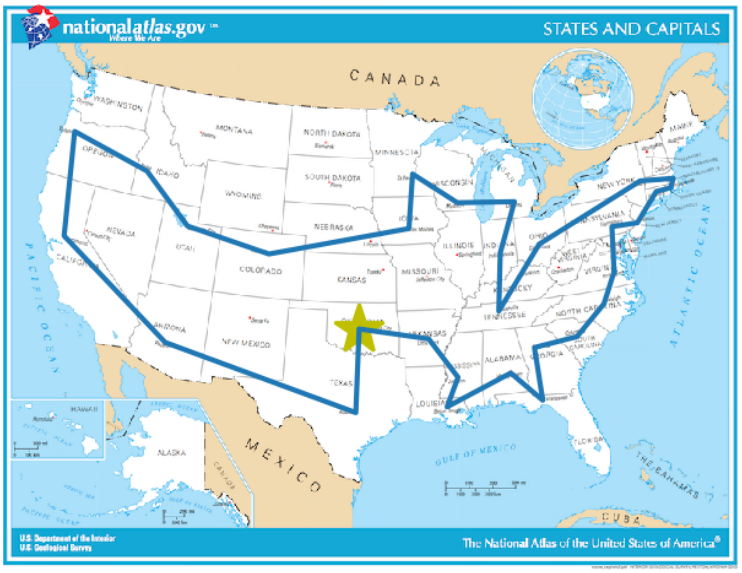
```
def successor(self, method='reverse'):
        if method == 'reverse':
            ind = sorted(random.sample(
                    [i for i,_ in enumerate(self.path)], 2))
            new_path = self.path[:]
            new_path = (new_path[:ind[0]] +
                        new_path[ind[0]:ind[1]][::-1] +
                        new_path[ind[1]:])
            return TravelingSalesmanProblem(new_path)
        elif method == 'permutation':
            new_path = self.path[:]
            random.shuffle(new_path)
            return TravelingSalesmanProblem(new_path)
        elif method == 'adjacent':
            successors = []
            for i in range(len(self.path)-1):
                new_problem = self.copy()
                new_problem.path[i], new_problem.path[i+1] =(
                new_problem.path[i+1], new_problem.path[i])
                successors.append(new_problem)

            last_path = self.copy()
            last_path.path[0], last_path.path[-1] =
last_path.path[-1], last_path.path[0]
            successors.append(last_path)
            return random.choice(successors)
        else:
            print('No valid method supplied')
            return False
```

We are almost ready to run the simulated annealing algorithm on the TSP. We first need to choose the decay rate and initial temperature in the scheduling function. Due to the exponential decay rate, the initial temperature does not matter too much, it just needs to be big enough. The parameter alpha dominates the shape of the temperature graph. If we choose alpha close to 1, the temperature would cool more slowly and the algorithm should take longer to run. I have empirically shown that the

solution is best when alpha is close to 1. I choose alpha=0.99 in this example. Putting everything into our main function, we have:

```
if __name__ == "__main__":
    num_cities = 30
    print('Solving the Simulated Annealing Approach')
    capitals_tsp = (
    TravelingSalesmanProblem(capitals_list[:num_cities]))
    starting_city = capitals_list[0]
    print("Initial path value: {:.2f}".format(-
capitals_tsp.fitness()))
    alpha = 0.99
    temperature=1e20
    result = simulated_annealing(capitals_tsp, Schedule(alpha,
temperature))
    print("Final path length: {:.2f}".format(result.fitness()))
    print(result.path)
    show_path(result.coords, starting_city, w=4, h=3)
```

If we run the script, we get a final path value of -2193.34 which is worse that our genetic algorithm has performed. I will do a more rigorous comparison in the next section. The resulting path looks like this:



Simulated annealing with 30 cities and alpha=0.99

Simulated Annealing vs. Genetic Algorithms

The physical differences between the two algorithms can be seen in the evolution of their successors. In GA's, a population is evolved at each

iteration in the attempt of mimicking darwinian evolution. In SA, a single successor is generated in an attempt to mimic annealing in metallurgy.

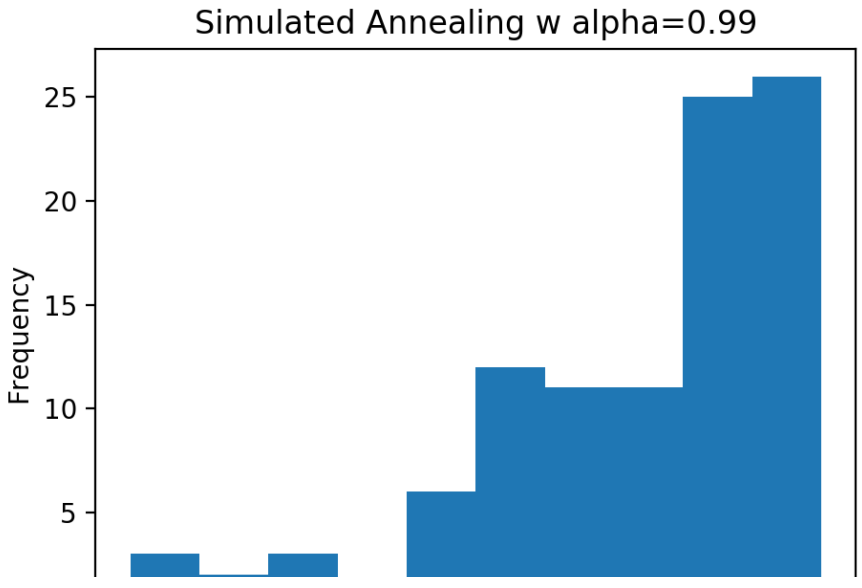
To compare the two algorithms, we need to consider how the algorithms handle candidate solutions whose value is considered worse than the current best solution. Rejecting all such solutions brings us back to a simple hill climbing algorithm. Sophisticated algorithms must temporarily accept worse solutions to get out of the local maxima.

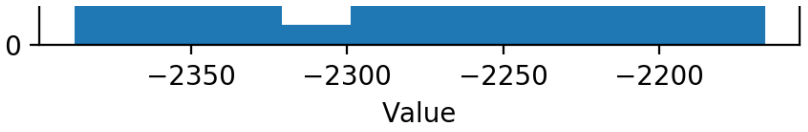
SA's accept worse solutions according to a probability that decreases as the temperature cools. Therefore, as the iteration number increases, the probability of accepting worse solutions decreases. The algorithms behaves like a random walk at the beginning and then slowly converges to the behavior of the simple hill climbing approach.

On the other hand, GA's generate an entire population of solutions at each iteration. The possibility of accepting worse solutions is encoded in the random selection parameter in the evolution function.

Even after drawing all these comparisons and differences, its not obvious to me which algorithm should perform better. GA's are clearly more costly since they need to generate whole populations. Let's move to an empirical comparison using our TSP problem as an example.

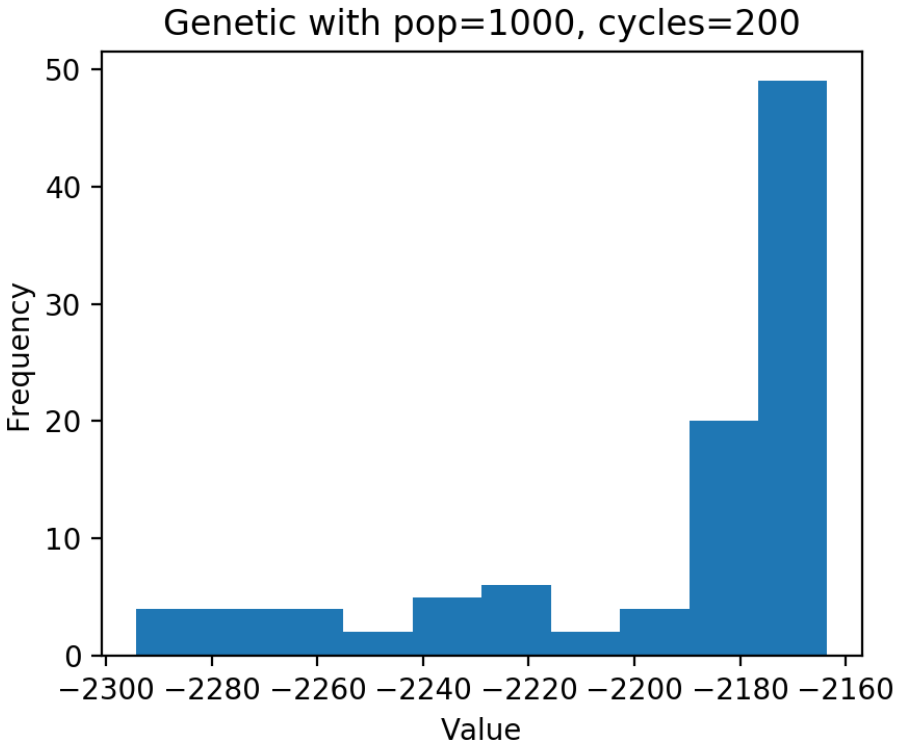
Let's run some simulations to determine which method looks better. If we run the simulated annealing algorithm 100 times (alpha=0.99), the distribution of solutions looks like:





100 runs of the simulated annealing algorithm with alpha=0.99. Best fitness was -2166.

Let's also run 100 iterations of the GA with a population size of 1000 with 200 evolution cycles. Note that the run time for this GA was about twice to that of the SA with alpha=0.99.



Monte Carlo for GA with 100 runs. Population size is 1000 and 200 evolution cycles.

Based on the histograms above, the GA algorithm is generally more stable and gives better results. Even if we increase alpha to 0.99999 in our SA, we cannot reach the GA minimal solution of -2163.47. The GA produces better values in the worst case scenario. It also produces better values in the best cases. The best solution found by the GA after the 100 Monte Carlo runs had a fitness of -2163.47. It looked like this:

Top performing genetic algorithm path with cost of -2163.4777.

The cost of my GA implementation could be greatly reduced using a multi-threaded approach. This is because individual search agents in an evolution cycle have no need to exchange messages. This would allow us to increase execution speed tremendously.

Conclusion

We have generated impressive solutions to a NP-hard problem using two probabilistic meta-heuristics. My empirical research has shown that GA nearly always beats SA, but at a higher computational cost. That said, a multi-threaded approach would greatly reduce the cost of GA. My recommendation is try both approaches if possible and determine for yourself which gives a better result.

If you've made it this far, drop me a line below — please let me know what you think! Feel free to add me on LinkedIn here.

References:

[1]https://en.wikipedia.org/wiki/Travelling_salesman_problem

[2]http://www.stat.yale.edu/~pollard/Courses/251.spring2013/Handouts/Chang-MoreMC.pdf — PAGE 64

[3]http://what-when-how.com/artificial-intelligence/a-comparison-of-cooling-schedules-for-simulated-annealing-artificial-intelligence/

[4]https://stackoverflow.com/questions/4092774/genetic-algorithm-versus-simulated-annealing-performance-comparison-and-use-cas

Ai Weekly Newsletter

Get a summary of our best articles each week.

First Name | Last Name

Email

Sign up

Machine Learning Artificial Intelligence Probability Computer Science Genetics

Discover Medium

Welcome to a place where words matter. On Medium, smart voices and original ideas take center stage - with no ads in sight. Watch

Make Medium yours

Follow all the topics you care about, and we'll deliver the best stories for you to your homepage and inbox. Explore

Become a member

Get unlimited access to the best stories on Medium — and support writers while you're at it. Just $5/month. Upgrade

About Help Legal