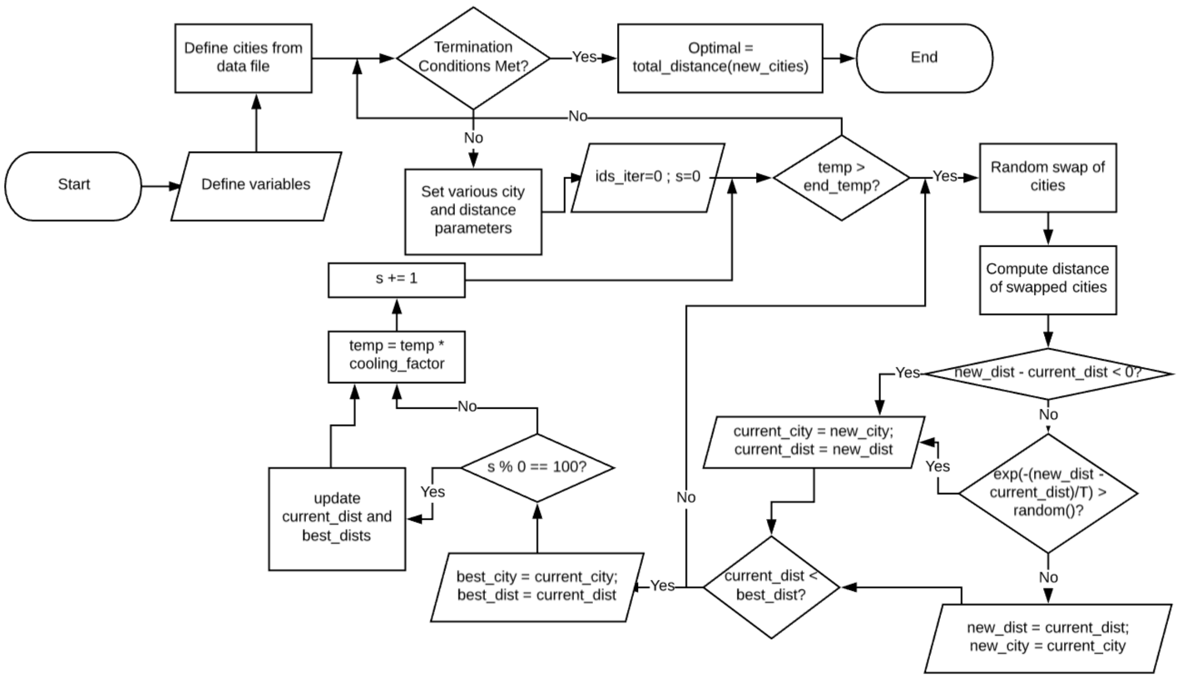
Rebecca Wood
Due: 2/11/2018

Simulated Annealing – Traveling Salesperson Problem

```
input: begin_temperature, end_temperature, cooling_factor, iterations, data_file
output: optimal_distance
import necessary packages

filename = read file for traveling salesperson problem
paired_distance = []
def read_file (filename):
	# read data file allowing cities = (name, index, city, latitude, longitude) for
	each line
	return cities
class City:
	def initialize (self, components in cities):
		# initialize all of the components by using self.component = component
def compute_distance_miles (self, city):
	# change latitude and longitude values to radians
	# use Haversine function to determine distance between cities in miles
	return function
def distance_miles (self, city):
	# call the paired_distances variable
	for origin_city in cities:
		# append paired_distances with these cities
		for destination in cities[: origin_city.index]:
			# for each pair compute the distance in miles from origin city to
			destination
def total_distance (cities):
	distances = distance_miles(city_A, city_B)
	return sum(distances)
def compute_swap_ind (index, nb_cities):
	# allow cities to swap places
	return (ind_prev, ind_next)
def distance_swap (cities, ind_a, ind_b):
	distances = []
	swap = compute_swap_ind(city_A, city_B)
	distance = distance_miles(swap)
	# append distances with distance, but avoid counting distances more than once
	return sum(distances)
def annealing (cities, begin_temperature, end_temperature, cooling_factor, iterations):
	# initialize best_cities, best_distances, current_distances, best_distance,
	ids_iterations
	while termination conditions not met:
		for i in range(iterations):
			# define temperature, current_cities, current_distance,
			new_distance, new_cities, step
			while end_temperature not met:
```

```
                                        # allow random swaps without swapping the first city
                                        # recompute only changed distances
                                        # allow the swap to reset and update new_cities
                                        best_cities = current_cities[:]
                                        best_distance = current_distance
                                        current_distances = update(current_distances)
                                        best_distances = update(best_distances)
                                        ids_iterations = update(ids_iterations)
                    optimal_distance = total_distance(new_cities)
          return optimal_distance
```

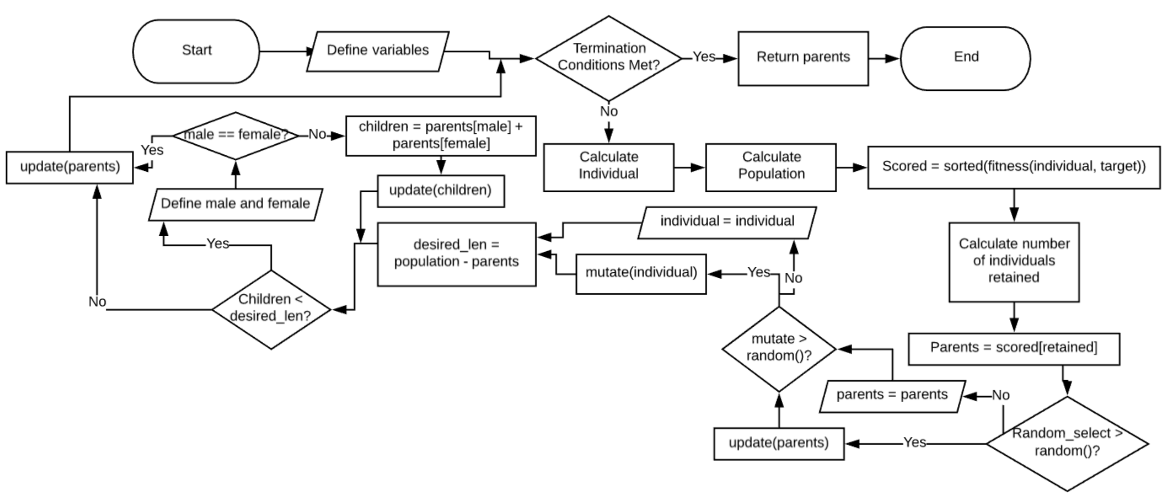


Genetic Algorithm – Generic Problem

```
input: length, min, max, population_max, target, retention rate, random_select, mutation
output: parents
import necessary packages

def individual (length, min, max):
          # return random integer values between min and max, of length = length
def population (count, length, min, max):
          pop = individual(length, min, max) for x in range(count)
          return pop
def fitness (individual, target):
          # total the values within individual
          return abs(target - total)
def grade (pop, target):
```

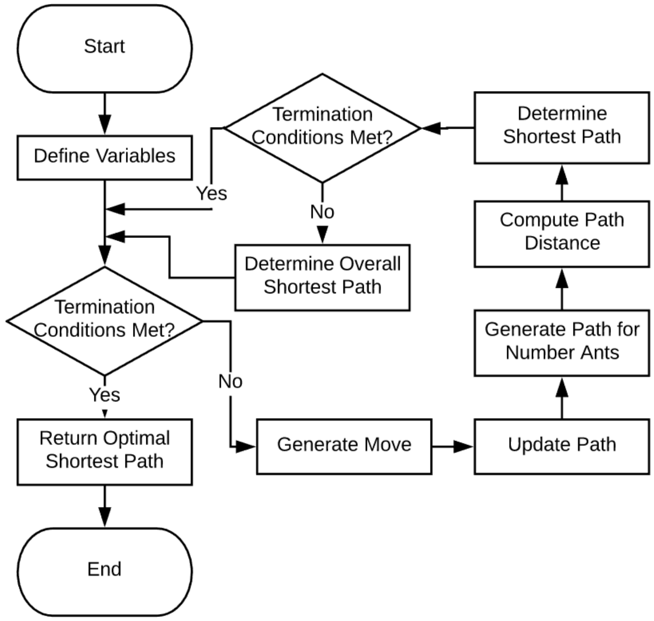
```
        # for each population total fitness(x, target)
        return total_fitness / len(pop)
def evolve (pop, target, retention rate, random_selection, mutation)
        scored = fitness(individual, target) for each individual in the population, sorted
        retain_len = length of scored * retention rate
        parents = individuals that are included in the scored list for length of
        retain_len
        for individual in parents:
                if random_select > random():
                        update(parents)
        for individual in parents:
                if mutate > random():
                        mutate_position = generateRandomInteger(0, len(individual) - 1)
                        # update(individual) with a random integer between the minimum
                        individual value and the maximum individual value
        # desired length is the difference between size of population and parents
        children = []
        while number of children is less than the desired length:
                male = generateRandomInteger(0, number of parents - 1)
                female = generateRandomInteger(0, number of parents - 1)
                if male is not equal to female:
                        male = parents[male]
                        female = parents[female]
                        child = half of the male and half of the female
                        update(children)
        update(parents) to include children
        return parents
```



Ant Colony Optimization – Shortest Path Problem

```
input: distances, number of ants, best ants, number of iterations, pheromone decay,
        pheromone weight (alpha), distance weight (beta)
output: overall shortest path
import necessary packages
```

```
class Colony:
	def initialize (self, distances, num_ants, best, iterations, decay, alpha, beta):
		# initialize variables such that self.variable = variable
		# define pheromone as distance / length of distances vector and index as a
		range of 0 to the length of distances vector
	def run (self):
		# define shortest path and overall shortest path
		for i in range(iterations):
			# define paths using generate paths function
			# allow pheromone spread to be based on the paths, best, and
			shortest paths components
			# update(shortest path) by the minimum of the paths
			if shortest path conditions are met:
				update(overall shortest path)
			update(pheromone) with decay value
		return overall shortest path
	def pheromone_spread (self, paths, best, shortest path):
		paths = sorted(paths)
		for path, distance in paths over the length of best:
			for move in path:
				updated(pheromone) for that particle move
	def generate_distance (self, path):
		# total distances for each move in paths
		return total
	def generate_paths (self):
		# initialize paths
		for i in range(num_ants):
			path = generate_path(starting location)
			update(paths) with (path, generate_distance(path))
		return paths
	def generate_path (self, starting location):
		# initialize path
		# define visited paths, starting with the starting location
		prev = starting location
		for i in range( len(distances) - 1 ):
			move = pick_move(pheromone[prev], distances[prev], visited paths)
			update(path) with (prev, move)
			prev = move
			update(visited paths) with move
		update(path) with (prev, start)
		return path
	def pick_move (self, pheromone, dist, visited):
		pheromone = copy(pheromone)
		pheromone[ list(visited) ] = 0
		row = pheromone ** alpha * ((1/dist) ** beta)
		norm_row = row / row.sum()
		move = choice( index, 1, norm_row)[0]
		return move
```

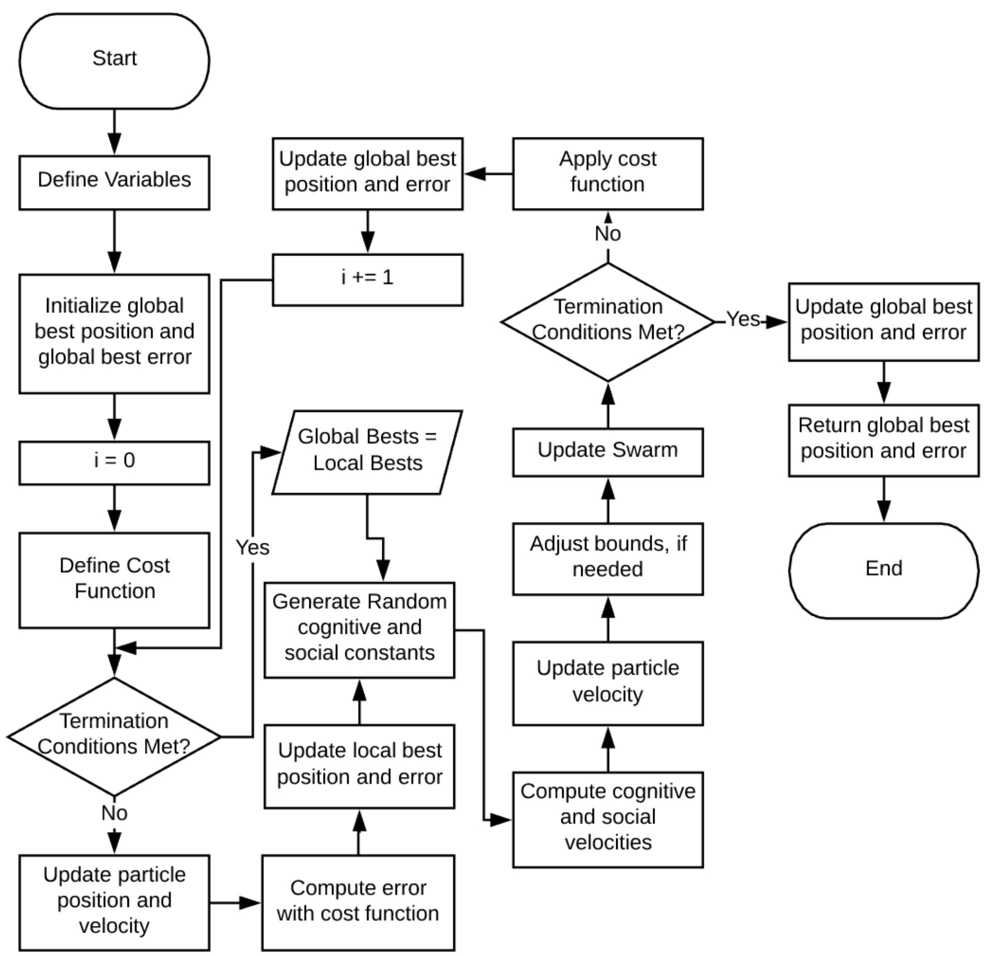


Particle Swarm Optimization – Generic Problem

```
input: initial position, cost function, bounds, number of particles, max
output: global best position, global best error
import necessary packages

def cost(x):
        # define cost function equation, returning a total value
class Particle:
        def initialize (self, initial position):
                # define particle position, particle velocity, local best position, local
                best error, local error
                for i in range(0, iterations):
                        # update(particle velocity) with a random velocity
                        # update(particle position)
        def fitness (self, cost function):
                # change error to the cost function as a function of particle position
                if error conditions are less than the best conditions:
                        # allow particle position and error to be classified as the best
        def velocity_func (self, global best position):
                # initialize weight, cognitive constant, and social constant
                for i in range(0, iterations):
                        # generate random values for r1 and r2
                        # compute cognitive and social velocity using the associated
                        constant, r-value, best position (local for cognitive and global for
                        social), and particle position
                        # update velocity with the weight, cognitive velocity and social
                        velocity
        def position_func (self, bounds):
```

```
        for i in range(0, iterations):
                # update(particle position)
                # adjust particle bounds if conditions are met
def initialize (self, cost func., initial position, bounds, num_particles, max):
        update(iterations)
        # define global best error and global best position
        swarm = []
        for i in range(0, num_particles):
                # update(swarm) with Particle(initial position)
        i = 0
        while termination conditions are not met:
                for j in range(0, number of particles):
                        # use cost function to apply fitness to swarm
                        if error conditions are met:
                                # update the global best error and position
                for j in range(0, num_particles):
                        # update(global best position) for swarm using the velocity
                        function
                        # update(position bounds) for swarm
                i += 1
        return global best position, global best error
```



Sources:

- http://toddwschneider.com/posts/traveling-salesman-with-simulated-annealing-r-and-shiny/
- https://lethain.com/genetic-algorithms-cool-name-damn-simple/
- https://www.codeproject.com/Articles/1104747/Introduction-to-Genetic-Algorithms-with-Python-Hel
- https://github.com/maoaiz/tsp-genetic-python/blob/master/tsp-genetic-python.py
- http://www.theprojectspot.com/tutorial-post/applying-a-genetic-algorithm-to-the-travelling-salesman-problem/5
- https://github.com/marcoscastro/tsp_pso
- https://nathanrooy.github.io/posts/2016-08-17/simple-particle-swarm-optimization-with-python/
- https://github.com/Akavall/AntColonyOptimization/blob/master/ant_colony.py
- https://github.com/pjmattingly/ant-colony-optimization
- Blackboard PowerPoints and Class-Given Python Code