1. (a) I tested my implementation of the simulated annealing algorithm on the 0/1 knapsack problem. My implemenation for the knapsack problem can be observed in the file **test_functions.py** and is included below for convenience.

```
class KnapsackProblem:

def __init__(self, max_weight, values, weights):
    self._max_weight = max_weight
    self._values = values
    self._weights = weights

def get_profit(self, x):
    weight = sum([x[i] * self._weights[i] for i in
        range(len(self._weights))])
    if weight > self._max_weight:
        return 1e12
    profit = -1 * sum([x[i] * self._values[i] for i in
        range(len(self._values))])
    return profit
```

To generate lists of values and weights I created lists of random numbers and included the number of elements corresponding to the problem size (for example, if I was testing problem size 5 I would use the first 5 elements of values and and the first 5 elements of weights for the Knapsack problem). To keep a reasonable **max_weight** parameter I calculated the mean of the **weights** set below and used the equation

$$max\_weight = problem\_size * mean$$

. The mean in this case was 37. Here are the random values and weights used:

```
values = [360, 83, 59, 130, 431, 67, 230, 52, 93,
          125, 670, 892, 600, 38, 48, 147, 78, 256,
          63, 17, 120, 164, 432, 35, 92, 110, 22,
          42, 50, 323, 514, 28, 87, 73, 78, 15,
          26, 78, 210, 36, 85, 189, 274, 43, 33,
          10, 19, 389, 276, 312, 360, 83, 59, 130, 431, 67, 230, 52, 93,
          125, 670, 892, 600, 38, 48, 147, 78, 256,
          63, 17, 120, 164, 432, 35, 92, 110, 22,
          42, 50, 323, 514, 28, 87, 73, 78, 15,
          26, 78, 210, 36, 85, 189, 274, 43, 33,
          10, 19, 389, 276, 312, 360, 83, 59, 130, 431, 67, 230, 52, 93,
          125, 670, 892, 600, 38, 48, 147, 78, 256,
          63, 17, 120, 164, 432, 35, 92, 110, 22,
          42, 50, 323, 514, 28, 87, 73, 78, 15,
          26, 78, 210, 36, 85, 189, 274, 43, 33,
          10, 19, 389, 276, 312, 360, 83, 59, 130, 431, 67, 230, 52, 93,
          125, 670, 892, 600, 38, 48, 147, 78, 256,
          63, 17, 120, 164, 432, 35, 92, 110, 22,
          42, 50, 323, 514, 28, 87, 73, 78, 15,
          26, 78, 210, 36, 85, 189, 274, 43, 33,
          10, 19, 389, 276, 312]

weights = [7, 0, 30, 22, 80, 94, 11, 81, 70,
           64, 59, 18, 0, 36, 3, 8, 15, 42,
```

```
9, 0, 42, 47, 52, 32, 26, 48, 55,
6, 29, 84, 2, 4, 18, 56, 7, 29,
93, 44, 71, 3, 86, 66, 31, 65, 0,
79, 20, 65, 52, 13, 7, 0, 30, 22, 80, 94, 11, 81, 70,
64, 59, 18, 0, 36, 3, 8, 15, 42,
9, 0, 42, 47, 52, 32, 26, 48, 55,
6, 29, 84, 2, 4, 18, 56, 7, 29,
93, 44, 71, 3, 86, 66, 31, 65, 0,
79, 20, 65, 52, 13, 7, 0, 30, 22, 80, 94, 11, 81, 70,
64, 59, 18, 0, 36, 3, 8, 15, 42,
9, 0, 42, 47, 52, 32, 26, 48, 55,
6, 29, 84, 2, 4, 18, 56, 7, 29,
93, 44, 71, 3, 86, 66, 31, 65, 0,
79, 20, 65, 52, 13, 7, 0, 30, 22, 80, 94, 11, 81, 70,
64, 59, 18, 0, 36, 3, 8, 15, 42,
9, 0, 42, 47, 52, 32, 26, 48, 55,
6, 29, 84, 2, 4, 18, 56, 7, 29,
93, 44, 71, 3, 86, 66, 31, 65, 0,
79, 20, 65, 52, 13]
```

I ran my algorithm on problem sizes ranging from 5 to 120 incrementing by 5 each time (5, 10, 15, 20, ..., 125). Each problem size involved running the algorithm 5 times and collecting statistics from those 5 runs in the same form as HW1 (mean $\pm$ std. dev). I chose to use 5 runs because the algorithm run time at high input sizes was getting to be too long. I received consistent profit values for all runs as seen in the data below.

My algorithm was run using the following parameters:

i. Starting temperature: 25000
ii. Final temperature: 0.1
iii. Number of iterations: 5
iv. Number of cycles: 20
v. Temperature reduction factor: 0.5
vi. Initial step value: 10 (insignifcant for discrete case)
vii. Step reduction factor: 0.9 (insignificant for discrete case)
viii. Initial point: (randomly 1 or 0)

```
[round(random.uniform(0, 1)) for _ in range(problem_size)]
```

ix. Allowed: Element of the set $\{0, 1\}$

Results:

Problem Size	Time (ms)	Profit
5	$1.16e+02 \pm 1.73e+00$	$-1.06e+03 \pm 0.00e+00$
10	$3.24e+02 \pm 3.30e+00$	$-1.56e+03 \pm 0.00e+00$
15	$6.31e+02 \pm 4.16e+00$	$-3.84e+03 \pm 0.00e+00$
20	$1.05e+03 \pm 4.72e+00$	$-4.44e+03 \pm 0.00e+00$
25	$1.55e+03 \pm 1.31e+01$	$-5.28e+03 \pm 0.00e+00$
30	$2.15e+03 \pm 1.37e+01$	$-5.83e+03 \pm 0.00e+00$
35	$2.84e+03 \pm 2.02e+01$	$-6.61e+03 \pm 0.00e+00$
40	$3.64e+03 \pm 1.68e+01$	$-6.97e+03 \pm 0.00e+00$
45	$4.53e+03 \pm 2.42e+01$	$-7.60e+03 \pm 0.00e+00$
50	$5.48e+03 \pm 1.03e+01$	$-8.59e+03 \pm 0.00e+00$
55	$6.59e+03 \pm 2.10e+01$	$-9.67e+03 \pm 0.00e+00$
60	$7.80e+03 \pm 3.71e+01$	$-1.02e+04 \pm 0.00e+00$
65	$9.09e+03 \pm 4.28e+01$	$-1.25e+04 \pm 0.00e+00$
70	$1.04e+04 \pm 1.35e+01$	$-1.30e+04 \pm 0.00e+00$
75	$1.21e+04 \pm 2.22e+02$	$-1.39e+04 \pm 0.00e+00$
80	$1.36e+04 \pm 9.38e+01$	$-1.44e+04 \pm 0.00e+00$
85	$1.52e+04 \pm 1.15e+02$	$-1.52e+04 \pm 0.00e+00$
90	$1.71e+04 \pm 2.48e+02$	$-1.56e+04 \pm 0.00e+00$
95	$1.89e+04 \pm 8.59e+01$	$-1.62e+04 \pm 0.00e+00$
100	$2.10e+04 \pm 3.81e+02$	$-1.72e+04 \pm 0.00e+00$
105	$2.31e+04 \pm 2.83e+02$	$-1.83e+04 \pm 0.00e+00$
110	$2.64e+04 \pm 2.07e+02$	$-1.88e+04 \pm 0.00e+00$
115	$2.87e+04 \pm 1.47e+02$	$-2.11e+04 \pm 0.00e+00$
120	$2.99e+04 \pm 8.68e+01$	$-2.16e+04 \pm 0.00e+00$

Analysis: My algorithm seemed to give very consistent results at the cost of high runtime. I believe this is because I used 20 as the default value for **num_cycles**, which reulted in a high amount of space exploration. I don't think this strategy would be feasible for very large problem sizes. I would like to test on large problem sizes (1000+) with a little more time to set up multithreading on a powerful server.

(b) I used the graph coloring problem to test multiple discrete values. More specifically, I used the Petersen graph which has 15 vertices and 10 nodes. Some nodes have 2 neighbors so the fewest number of colors that will satisfy the problem is 3. I wrote the following code for testing:

```python
class GraphColoring:

   def __init__(self, colors):
      self._graph = nx.petersen_graph()
      self._colors = colors
      self._color_map = { node : random.choice(self._colors) for node in
          self._graph.nodes }

   def assign_color_map(self, proposed_colors):
      for i, key in enumerate(self._color_map.keys()):
         self._color_map[key] = proposed_colors[i]

   def _verify_graph(self):
      for node in self._graph.nodes:
         for neighbor in self._graph.neighbors(node):
```

```
            if self._color_map[node] == self._color_map[neighbor]:
               return 0
         return -1

      def solve(self, proposed_colors):
         self.assign_color_map(proposed_colors)
         return self._verify_graph()

      def print_colors(self):
         for node in self._graph.nodes:
            print("Node:")
            print("\t[{0}] -> [{1}]".format(node, self._color_map[node]))
            print("Neighbors:")
            for neighbor in self._graph.neighbors(node):
               print("\t[{0}] -> [{1}]".format(neighbor,
                  self._color_map[neighbor]))
```

I ran the algorithm 100 times with simulated annealing parameters similar to those in problem 1.a. The code for the test is in **driver.py**. The x0 parameter involved a list of random colors constructed using the following code:

```
color_set = range(3)
[random.choice(color_set) for _ in range(problem_size)]
```

The following data was pulled from the 100 runs:

Time (ms)	Solved (-1 or 0)
$2.71e+02 \pm 2.00e+01$	$-1.00e+00 \pm 0.00e+00$

Analysis: The 10-node graph coloring problem was solved correctly every time given a random input list. The average runtime of 271 milliseconds seemed reasonable.

2. I implemented the Nelder-Mead algorithm. The implementation can be found in **nelder_mead.py**. I compared against the continuous version of the simulated annealing algorithm by minizing the following function 1000 times:

$$F(x_0, x_1) = x_0^2 + 2x_1^2 + 2x_0x_1$$

$$Mininmum = 0, 0$$

$$F(min) = 0$$

Not the most exciting function, but the Nelder-Mead algorithm took a long time to implement so I did not have as much time to play around with it. The parameters for the simulated annealing algorithm were the same as the previous problems. The x0 parameter was generated using a uniform distribution from -100 to 100 using the following Python code:

```
(random.uniform(-100, 100), random.uniform(-100, 100))
```

The following statistics were gathered for the Nelder-Mead algorithm:

Time (ms)	Minimum
$3.39e+00 \pm 1.14e+00$	$1.87e-15 \pm 2.36e-15$

And the following stats were gathered for the continuous simulated annealing function:

Time (ms)	Minimum
$2.13e+01 \pm 4.61e+00$	$1.78e-03 \pm 2.87e-03$

It looks like the Nelder-Mead algorithm performed better with respect to both time and minimum accuracy for the given problem. The simulated annealing algorithm also has a high amount of variance for real valued functions.