Assignment 1 Q2-1_AG03

September 28, 2022

Group Name: AG 03.

Student Name (Student ID):

1. LAU Zhe Ru Zachary (A0256006M)
2. ONG Ee Wen Lennard (A0034832L)
3. Jeremy LIAN Zhi Wei (A0250972B)

1 Question 2

1.1 Introduction to question 2

In the second question of this assignment, we will explore the use of local search in genome assembly.

We will use local search to assemble (construct) a large part of the nucleotide sequence of the monkeypox virus, which has been downloaded from the National Center for Biotechnology Information in the United States. Please note that no additional or specialized knowledge of biology or bioinformatics is required for this assignment. (Actually, the technical specifics of bioinformatics have been adapted and simplified for the purposes of this computer science assignment, so if you are a biologist, please do not apply preexisting knowledge to solve the problem. Furthermore, you should not attempt to search up the genome on genomic databases to "guess" the actual sequence, since we are more interested in your coding methodology rather than your attempts at reproducing a known sequence.)

This is an introductory computer science assignment and not a bioinformatics assignment; we are simply using bioinformatics as a use case to illustrate the applicability of local search to the natural sciences. Therefore, no knowledge of bioinformatics is assumed or required. In the paragraphs that follow, I will give a short crash course which will cover all the domain knowledge you will need to know in order to tackle this problem.

For technical reasons, when we analyze the nucleotide sequence (genome) of a virus, we usually cannot "read" it in one fell swoop. We have to read the genome in parts, because the genome is usually too long for the machine to read in a single sitting. To simplify things, a "read" is a single view of part of the genome; think of it as a SUBSTRING, a partial view of the whole genome. After we have generated multiple reads of a genome, we then have to "stitch", or combine, the different reads of the genome together. This process of stitching up reads of a genome into the final sequence is known as genome assembly. However, the different reads of the genome cannot just be concatenated like usual string concatenation. It's not a situation where you have one read, "Hello", and another read, "World", and all you need to do is concatenate both strings together to make "Hello World". Among other reasons, there are two major reasons why you can't do so:

1. You do not know which read came first. The reads are not ordered. How do you know "Hello" came after "World"? The answer is that you don't. Imagine how complicated this situation might be if you had more than two reads. (This is indeed our situation, where we have n reads, and $n >> 2$.)
2. One read may contain a substring contained in another read. Specifically, without loss of generality, part of the ending x characters of a read (i.e., suffix) might also be found in the starting x positions (i.e., prefix) of another read.

- A computer scientist usually creates opportunities from problems. While this may be a "problem" in that you just can't concatenate two strings blindly, the fact that strings contain shared "substrings" is actually a very helpful clue that you can use to "join" strings together.
- Note that the choice of the value of x could be a hyperparameter decided by the computer scientist.

1.2 Your tasks

In this part of the assignment, you will work with (simulated) reads that I have generated from the nucleotide sequence of the monkeypox virus. In reality, bioinformatics is far more complicated, but here we will work with a simplified situation. Your task is to examine the reads that I have provided for you, and from there "infer" the nucleotide sequence that might have produced those reads.

The reads are provided in the csv file `data.csv` which simply provides a list of unique strings. Note that you should NOT assume any particular ordering of the strings in this dataframe. In fact, the strings have already been shuffled randomly.

NOTE: You are not allowed to use `pandas` or any other libraries apart from the Python STL to load the csv file.

1.2.1 Task A (3 marks):

Create a directed graph. The nodes in the graph are the strings in the list of reads. An edge should be drawn FROM read A TO read B if and only if a suffix (of length x) of read A is also a prefix (obviously, also of length x) of read B. For the purposes of the assignment, limit the value of x to between 5 and 30, both inclusive. That is, to be clear, $5 \leq x \leq 30$. The weight of an edge between read A and read B should be the NEGATED value of x, i.e. $-x$.

In your Jupyter notebook, please report the number of edges in your graph. Provide a barplot or histogram which shows the number of edges with different weights or weight categories. In this task, you are free to use plotting libraries such as `matplotlib` or `seaborn` to plot this graph.

As an example, if read A is "TACTAGT" and read B is "TAGTCCCCT", then an edge is drawn FROM read A TO read B (i.e., $A \rightarrow B$) with weight of -4. This is because the 4-suffix "TAGT" is also the 4-prefix of read B; in other words, the last 4 characters of read A (a substring of length 4) overlap with the first 4 characters of read B (a substring of length 4).

1.2.2 Task B (7 marks):

From Task A, you now have a graph which shows connections between reads based on how they overlap, in theory you could draw a path through the graph and thereby derive the full sequence (genome).

Task B asks you to use local search method(s) to determine a path through this directed graph of strings.

- You are expected to use simulated annealing and tune the relevant configuration settings and hyperparameters. The minimum requirement is to implement simulated annealing.
- Explain tha rationale behind the choice of scheduling strategy and parameters.
- However, you may also explore other search methods in addition to simulated annealing. Marks will be awarded for effort.

Note the following constraints:

1. The path has to go through each and every vertex exactly once. For computer scientists, this constraint is reminiscent of the “Traveling Salesman’s Problem”, except that unlike TSP, we should not need to go back to the starting vertex again.
2. For the purposes of neighbor generation / action selection at each node, bear in mind that a path through the graph which minimizes the total number of nucleotides in the assembled sequence is the preferred path. To state that another way, the assembled sequence should be derived from a path that goes through EACH and EVERY vertex exactly once, however we want this assembled sequence to be AS SHORT AS POSSIBLE.
3. You are not given the starting (source/origin) or ending (destination) vertex.
4. For avoidance of ambiguity, no cycles are allowed. You must not visit a vertex more than once.
5. You are not allowed to use any libraries apart from the Python Standard Library. No import statements which import libraries outside of the Python STL should be found within your answer for Task B.

Please remember to report the assembled sequence that you obtain. Although it would be great if you can come up with a good sequence, please feel reassured that we are more interested in your APPROACH to the problem, and so you can potentially get a reasonable score on this task even if your solution is “wrong”. It is the process, rather than the result, which matters more.

2 NOTE:

A report summary of our work is included at the end, with figures generated from the code cells in this notebook. You may choose to run the code cells but due to large hyperparameters involved, some cells may take longer than 50 mins to run. Additionally, due to the stochastic nature of the algorithm used in this task, outputs may differ between what is reflected in our report and the cell blocks. Figures will be saved into the path from which you open this notebook.

3 Part 1 - Implementation of a graph structure of data.csv.

Input Data: to be structured into a dictionary {index:read_sequence}

3.0.1 Create Graph Object

A graph object which is a dictionary of dictionaries is created.

It follows the format:

{ A: {B: overlapScore, C: overlapScore, … } B: {D: overlapScore, F: overlapScore, … } C: {A: overlapScore, E: overlapScore, … } … }

[1]:
```python
import copy
from typing import Type

class Graph:
    """
    Taken from AIME4e, search.py

    A graph connects nodes (vertices) by edges (links). Each edge can also
    have a length associated with it.
    The constructor call is something like:
        g = Graph({'A': {'B': 1, 'C': 2})
    this makes a graph with 3 nodes, A, B, and C, with an edge of length 1 from
    A to B,  and an edge of length 2 from A to C. You can also do:
        g = Graph({'A': {'B': 1, 'C': 2}, directed=False)
    This makes an undirected graph, so inverse links are also added. The graph
    stays undirected; if you add more links with g.connect('B', 'C', 3), then
    inverse link is also added. You can use g.nodes() to get a list of nodes,
    g.get('A') to get a dict of links out of A, and g.get('A', 'B') to get the
    length of the link from A to B. 'Lengths' can actually be any object at
    all, and nodes can be any hashable object."""

    def __init__(self, graph_dict=None, directed=True):
        self.graph_dict = graph_dict or {}
        self.directed = directed
        if not directed:
            self.make_undirected()

    def make_undirected(self):
        """Make a digraph into an undirected graph by adding symmetric edges."""
        for a in list(self.graph_dict.keys()):
            for (b, dist) in self.graph_dict[a].items():
                self.connect1(b, a, dist)

    def connect(self, A, B, distance=1):
        """Add a link from A and B of given distance, and also add the inverse
        link if the graph is undirected."""
        self.connect1(A, B, distance)
        if not self.directed:
            self.connect1(B, A, distance)
    def connect1(self, A, B, distance):
        """Add a link from A to B of given distance, in one direction only."""
        self.graph_dict.setdefault(A, {})[B] = distance

    def get(self, a, b=None):
        """Return a link distance or a dict of {node: distance} entries.
        .get(a,b) returns the distance or None;
        .get(a) returns a dict of {node: distance} entries, possibly {}."""
        links = self.graph_dict.setdefault(a, {})
        if b is None:
            return links
        else:
            return links.get(b)

    def nodes(self):
        """Return a list of nodes in the graph."""
        s1 = set([k for k in self.graph_dict.keys()])
        s2 = set([k2 for v in self.graph_dict.values() for k2, v2 in v.items()])
        nodes = s1.union(s2)
        return list(nodes)

    def __repr__ (self):
        print (self.graph_dict)
```

3.0.2 Extract CSV

[2]:
```python
import csv
from pprint import pprint

# Import the data, discarding column 2 because its similar to column 1
data_reads = []
with open ('data.csv', 'r') as csv_file:
    for line in csv.reader(csv_file, delimiter=','):
        data_reads.append([line[0],line[2]])

data_reads = [[int(row[0]), row[1]] for row in data_reads[1:]] #discard the␣
 ↪header row, convert types
```

3.0.3 Generate Directed Graph

A function called overlap is created to generate a graph.

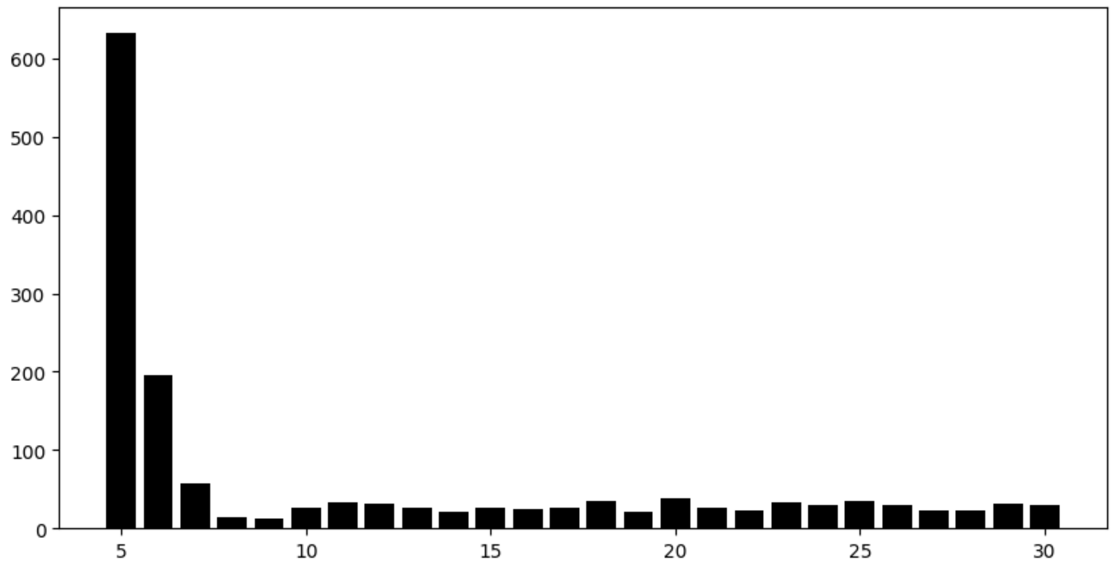
[3]:
```python
import copy

def overlap(readlist: list): # takes
    """Returns a directed graph of overlap scores for a genome list
    Inputs
    - readlist: the csv in list format [id: 'genome seq']
    - graph_output: a Graph() object instance
    """
    graph_output = Graph()
    list_check = copy.deepcopy(readlist)

    while list_check:
        line_check = list_check.pop()
        for num_characters in reversed(range(5,31)):
            idx_matched = []
            suffix = line_check[1][-num_characters:]
            for idx, line_reads in enumerate(readlist):
                prefix = line_reads[1][:num_characters]
                # print (f'CheckID{line_check[0]}, ReadID: {idx}, suffix:␣
 ↪{suffix}, prefix: {prefix}')
                if (suffix == prefix) and (idx not in idx_matched):
                    graph_output.connect(line_check[0], line_reads[0],␣
 ↪num_characters)
                    idx_matched.append(idx)

    return graph_output

LOOKUP_GRAPH = overlap(data_reads)
```

3.1 Plotting

Simple histogram to see distribution of overlap # and count

[4]:
```
import matplotlib.pyplot as plt
# matplotlib.rcParams.update(_VSCode_defaultMatplotlib_Params) #force␣
 ↪matplotlib to print white

histogram = {} #value: count

for node in LOOKUP_GRAPH.nodes():
    adjacencies = LOOKUP_GRAPH.get(node)
    for item, overlap in adjacencies.items():
        #check if present
        if histogram.get(overlap):
            histogram[overlap] += 1
        else:
            histogram[overlap] = 1

overlap_labels = list(histogram.keys())
overlap_counts = list(histogram.values())
```

```
fig = plt.figure(figsize = (10, 5))
plt.bar(overlap_labels, overlap_counts, color = 'black')
plt.show()
```



4 Part 2 - Implementation of Local Seach using Simulated Annealing

4.1 Problem Class

[5]:
```
class Problem(object):
    """The abstract class for a search problem."""

    def __init__(self, initial=None, goals=(), **additional_keywords):
        """Provide an initial state and optional goal states.
        A subclass can have additional keyword arguments."""
        self.initial = initial  # The initial state of the problem.
        self.goals = goals      # A collection of possible goal states.
        self.__dict__.update(**additional_keywords) ##DUNDER dict, any new␣
↪initializations go here

    def actions(self, state):
        """Return a list of actions executable in this state."""
        raise NotImplementedError # Override this!

    def result(self, state, action):
        "The state that results from executing this action in this state."
        raise NotImplementedError # Override this!
    def is_goal(self, state):
        "True if the state is a goal."
        return state in self.goals # Optionally override this!

    def action_cost(self, state, action, result=None):
        "The cost of taking this action from this state."
        return 1 # Override this if actions have different costs
```

4.2 TSP Subclass

[6]:
```
# Code to generate neighbours, value of states, etc.
import random

class TSP(Problem):
    #Implement TSP class here
    def __int__(self, initial: list = None, lookup_graph: type[Graph] = None,␣
 ↪swops: int = 1):
        """
        Inputs
        - swops - an integer indicating the number of pair swops
        - lookup_graph: a Graph() object instance that is related to `initial`
        - initial - a list indicating the sequence of steps through a problem
        """
        super().__init__(initial)
        self.lookup_graph = lookup_graph
        self.swops = swops

    def N_opt(self, state: list) -> list:
        '''Neighbour generating function for Traveling Salesman Problem

        Inputs
            List - the current node sequence as a list
        '''
        state2 = state[:]
        for _ in range(self.swops):
            l = random.randint(0, len(state2) - 1)
            r = random.randint(0, len(state2) - 1)
            if l > r:
                l, r = r,l
            state2[l : r + 1] = reversed(state2[l : r + 1])
        return state2

    def actions (self, state: list) -> list:
        """Returns a list of STATES that can follow from present state
        """
        return [self.N_opt]
    def result(self, state: list, action: list) -> list:
        """ Returns a list with new sequence through a graph
        """
        return action(state)

    def path_cost(self, state: list) -> int:
        '''Returns total overlapping score. In general, higher = better
        e.g. score 100 indicates a total of 100 characters have been overlapped.

        Inputs
            path_list - list of steps taken through graph
            lookup - overlap lookup graph of genome fragments
        '''
        res = 0
        for i in range(len(state) -1):
            n = state[i]
            n1 = state[i+1]
            score_word = self.lookup_graph.get(n, n1)
            if score_word is not None:
                res += score_word
        return res

    def value(self, state: list) -> int:
        """ Returns negative value of the path cost"""
        return -1 * self.path_cost(state)

tsp_problem = TSP(LOOKUP_GRAPH.nodes(), lookup_graph = LOOKUP_GRAPH, swops=500)
```

4.3 Node Class

Node class from AIMA. Extensions to allow expansion and path.

[7]:
```
# Use the following Node class to generate search tree
import math
class Node:
    """A node in a search tree. Contains a pointer to the parent (the node
    that this is a successor of) and to the actual state for this node. Note
    that if a state is arrived at by two paths, then there are two nodes with
    the same state. Also includes the action that got us to this state, and
    the total path_cost (also known as g) to reach the node. Other functions
    may add an f and h value; see best_first_graph_search and astar_search for
    an explanation of how the f and h values are handled. You will not need to
    subclass this class."""

    def __init__(self, state, parent=None, action=None, path_cost=0):
        """Create a search tree Node, derived from a parent by an action."""
        self.state = state
        self.parent = parent
        self.action = action
        self.path_cost = path_cost
        self.depth = 0
        if parent:
            self.depth = parent.depth + 1

    def __repr__(self):
        return "<Node {}>".format(self.state)

    def __lt__(self, node):
        return self.state < node.state

    def expand(self, problem):
        """List the nodes reachable in one step from this node."""
        return [self.child_node(problem, action)
                for action in problem.actions(self.state)]

    def child_node(self, problem, action):
        """[Figure 3.10]"""
        next_state = problem.result(self.state, action)
        next_node = Node(next_state, self, action, problem.path_cost(self.
↪state))
        return next_node

    def solution(self):
        """Return the sequence of actions to go from the root to this node."""
        return [node.action for node in self.path()[1:]]

    def path(self):
        """Return a list of nodes forming the path from the root to this node.
↪"""
        node, path_back = self, []
        while node:
            path_back.append(node)
            node = node.parent
        return list(reversed(path_back))
```

4.4 Simulated Annealing Definition

SA definition with a logging dictionary to trace the behaviour of the system within each experiment.

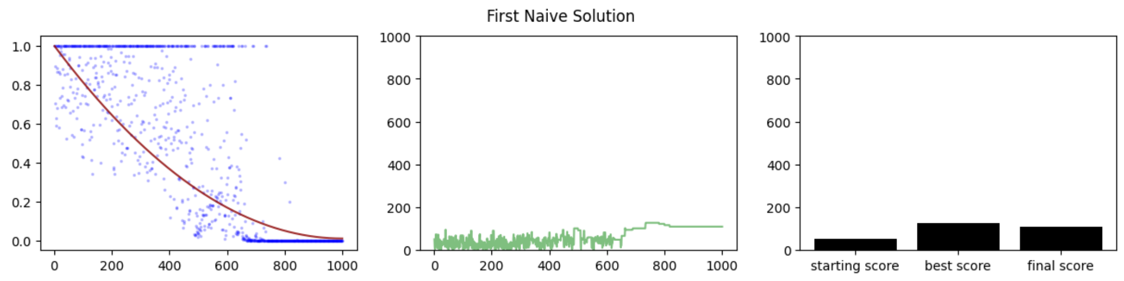
[8]:
```
import random

#HELPER FUNCTIONS
def probability(prob: float) -> bool:
    """ returns a bool based on probability

    Inputs:
        prob:  should be a number between 0 - 1.0
        e.g. `prob` = 0.25 -> 25% probability of returning TRUE
    """
    return random.uniform(0.0,1.0) < prob

def scheduler(stp_max: int = 1000, power: int = 1, tmp_max: float = 100.0,␣
 ↪tmp_min: float = 1.0) -> list:
    """ Returns a list of temperatures to be used

    Inputs:
        power : Nth power of Power-N curve
        stp_max: number of steps.
        tmp_max = max temperature. arbitrarily at 100
        tmp_min = min temperature. currently at 1.0
    """
    ''' Initialize '''
    stp_current = 1
    stp_max = stp_max + 1
    tmp_range = tmp_max - tmp_min
    tmp = []

    while stp_current < stp_max:
        '''Power-N Curve Cooling'''
        tmp_current = tmp_min + tmp_range * ((stp_max - stp_current) / stp_max)␣
 ↪** power

        '''update variables'''
        tmp.append(tmp_current)
        stp_current += 1
    return tmp


def simulated_annealing_full(problem, sch_steps: int = 1000, sch_power: int =␣
 ↪2, n_swops: int = 1):
    solution_tree_current = Node(problem.initial) #initialize the solution tree
    temp_schedule = scheduler(stp_max = sch_steps, power = sch_power)
    state_log = []
    step = 0

    #create logging
    log = {}
    log['temp'] = []
    log['accp'] = []
    log['delt'] = []
    log['step'] = []
    log['scor'] = []
    log['prob'] = []
    log['state'] = []

    """Run simulated annealing process"""
    while temp_schedule:
        temp = temp_schedule.pop(0)

        """Generate next and evaluate"""
        solution_tree_next = random.choice(solution_tree_current.
 ↪expand(problem))
        energy_current = problem.value(solution_tree_current.state)
        energy_next = problem.value(solution_tree_next.state)
        energy_delta = energy_current - energy_next #recall, this is flipped

        if energy_delta < 0:
            probablity_score = (math.e ** (energy_delta / temp))
            accept = probability(probablity_score)
        else:
            probablity_score = 1
            accept = True

        """Update for next cycle"""
        if accept:
            solution_tree_current = solution_tree_next

        """LOGGING"""
        step += 1
        state_log.append(solution_tree_current.state)
        log['temp'].append(temp)
        log['accp'].append(accept)
        log['delt'].append(energy_delta)
        log['step'].append(step)
        log['prob'].append(probablity_score)
        log['scor'].append(problem.path_cost(solution_tree_current.state))
        log['state'].append(solution_tree_current.state)

    return log

states = simulated_annealing_full(tsp_problem)
```

4.5 Graphing Function

Helper function for composite scatter plot and linegraphs to observe the interplay of temperature, acceptance rate and its impact on the score.

```
[9]: # Helper function to plot
def plot_results(states: dict = states, filename: str = 'default', title: str =␣
 ↪'default', show = True, save = True):
    ''' Plots the results of search algoritm

    Inputs:
        state = final state
        filename = filename to save as
    '''
    fig, ax = plt.subplots(nrows = 1, ncols = 3, figsize=(15,3))

    # Preparing the data to subplots
    x = states.get('step')
    stats = {}
    stats['starting score'] = (states.get('scor')[0]) #starting score
    stats['best score'] = max(states.get('scor')) #min score
    stats['final score'] = states.get('scor')[-1] #end score

    #setting plots

    ax[0].plot(x, states['prob'], linestyle = 'none', marker = 'o', color =␣
 ↪'blue', markersize = 1.5, alpha = 0.2)
    ax[0].plot(x, list(map(lambda temp: temp / 100, states['temp'])), color =␣
 ↪'darkred', alpha = 0.8)

    ax[1].set_ylim(0,1000)
    ax[1].plot(x, states['scor'], linestyle = 'solid', color = 'green', alpha =␣
 ↪0.5)

    ax[2].set_ylim(0,1000)
    ax[2].bar(stats.keys(), stats.values(), color = 'black')


    # ax[1].scatter(x, states['scor'])

    # Title
    plt.suptitle(f'{title}')
    if save:
        plt.savefig('img/' + filename)
    if show:
        plt.show()
    plt.close()

"""
Debugging and Testing
"""
```

```
plot_results(states, filename = 'test1', title = 'First Naive Solution', show =␣
↪True)
```



4.6 Figure Explanation

1) First graph on the left shows probablity of bad neighbour state acceptance with iteration steps. The temperature decay function is overlayed in red.
2) Second graph shows the overlap scores of the current state at each iteration step.
3) Third graph compares the initial state score, the best score achieved during the course of iteration, and the final arrived solution score.

5 Exercise

5.1 Exploring hyperparameters

For hyperparameter exploration, the interaction of 3 hyperparameters are explored: 1. Step Range: the total number of steps in the simulation 2. Decay Power: the rate of temperature decay, expressed as a N-power curve 3. Number of Swops: the number of pair-swops per neighbour-generation function.

[10]:
```
"""Defining Hyperparameters to Explore"""
step_range = [100, 1000, 10000, 100000]
decay_power = [0.1, 1, 10, 100]
num_swops = [1, 10, 100]

"""Setup experiment logging"""
save_prefix = '1100'
experiment_log = {} # store data in the format: {swopX : {a: [1,2,3], b: [1,2,3.
↪..],...}, swop : ....} where swop is num_swops and a, b, c are parameters in␣
↪each run

"""Run Experiment"""
for swop in num_swops:
    # Setting up model
    tsp_problem = TSP(LOOKUP_GRAPH.nodes(), lookup_graph = LOOKUP_GRAPH, swops␣
↪= swop)
    # states = simulated_annealing_full(tsp_problem)
    for steps in step_range:
        for power in decay_power:
            #run simulation
            states = simulated_annealing_full(tsp_problem, sch_steps = steps,␣
↪sch_power = power)

            #extract data
            filename = str(f'[{save_prefix}]␣
↪graphs_swop{swop}_decay{power}_steps{steps}')
            title = str(f'swops:{swop}, decay_power:{power}, steps: {steps} |␣
↪best_v_final_score: {max(states.get("scor"))}, {states.get("scor")[-1]}')

            #plot results
            print(title)
            plot_results(states = states, filename = filename.replace(".",␣
↪"pt"), title = title.replace(".", "pt"), show = False, save = True)

            # Log results
            key = str(f'swop{swop}_decay{power}_steps{steps}')
            experiment_log[key] = states
```

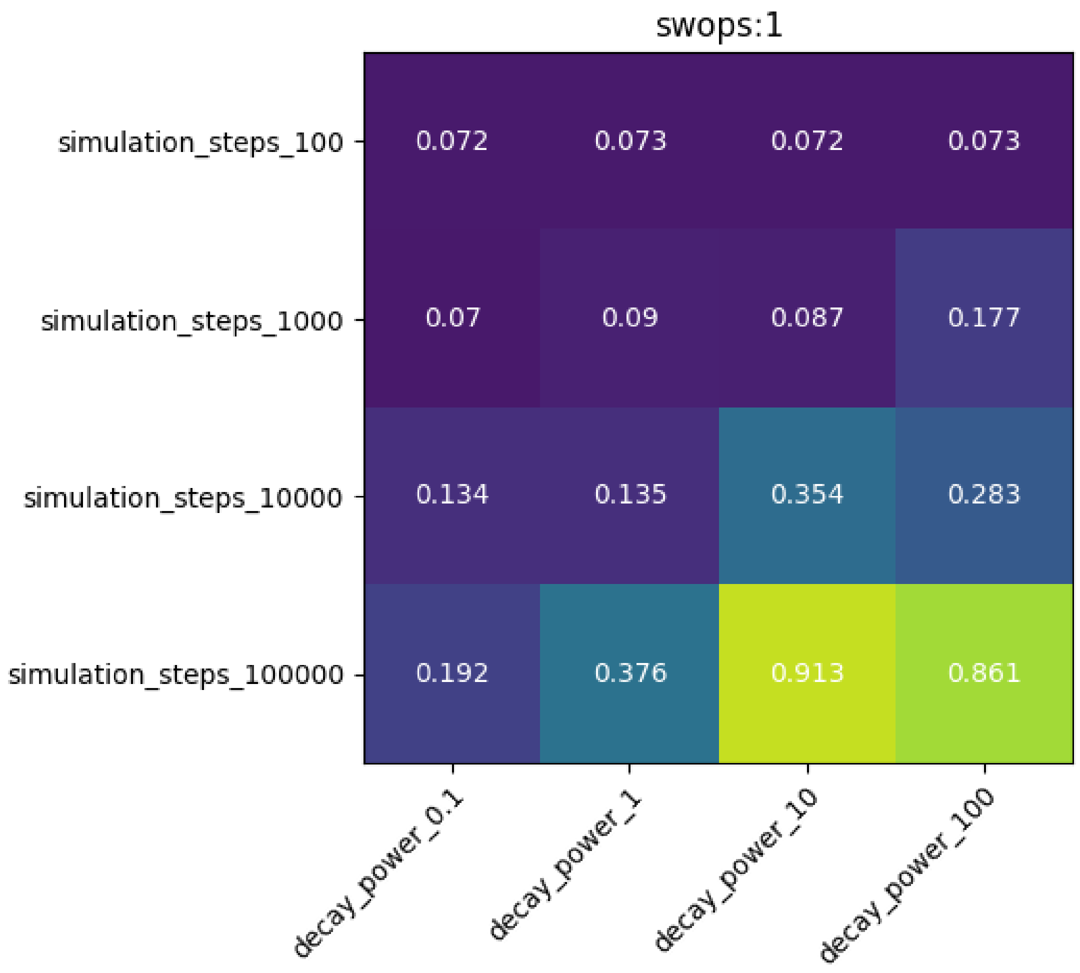
```
swops:1, decay_power:0.1, steps: 100 | best_v_final_score: 72, 11
swops:1, decay_power:1, steps: 100 | best_v_final_score: 73, 72
swops:1, decay_power:10, steps: 100 | best_v_final_score: 72, 71
swops:1, decay_power:100, steps: 100 | best_v_final_score: 73, 73
swops:1, decay_power:0.1, steps: 1000 | best_v_final_score: 70, 20
swops:1, decay_power:1, steps: 1000 | best_v_final_score: 90, 85
swops:1, decay_power:10, steps: 1000 | best_v_final_score: 87, 87
swops:1, decay_power:100, steps: 1000 | best_v_final_score: 177, 177
swops:1, decay_power:0.1, steps: 10000 | best_v_final_score: 134, 20
swops:1, decay_power:1, steps: 10000 | best_v_final_score: 135, 100
swops:1, decay_power:10, steps: 10000 | best_v_final_score: 354, 354
swops:1, decay_power:100, steps: 10000 | best_v_final_score: 283, 283
swops:1, decay_power:0.1, steps: 100000 | best_v_final_score: 192, 10
swops:1, decay_power:1, steps: 100000 | best_v_final_score: 376, 376
swops:1, decay_power:10, steps: 100000 | best_v_final_score: 913, 913
swops:1, decay_power:100, steps: 100000 | best_v_final_score: 861, 861
swops:10, decay_power:0.1, steps: 100 | best_v_final_score: 152, 28
swops:10, decay_power:1, steps: 100 | best_v_final_score: 73, 73
swops:10, decay_power:10, steps: 100 | best_v_final_score: 93, 93
swops:10, decay_power:100, steps: 100 | best_v_final_score: 75, 75
swops:10, decay_power:0.1, steps: 1000 | best_v_final_score: 123, 10
swops:10, decay_power:1, steps: 1000 | best_v_final_score: 208, 208
swops:10, decay_power:10, steps: 1000 | best_v_final_score: 172, 172
```

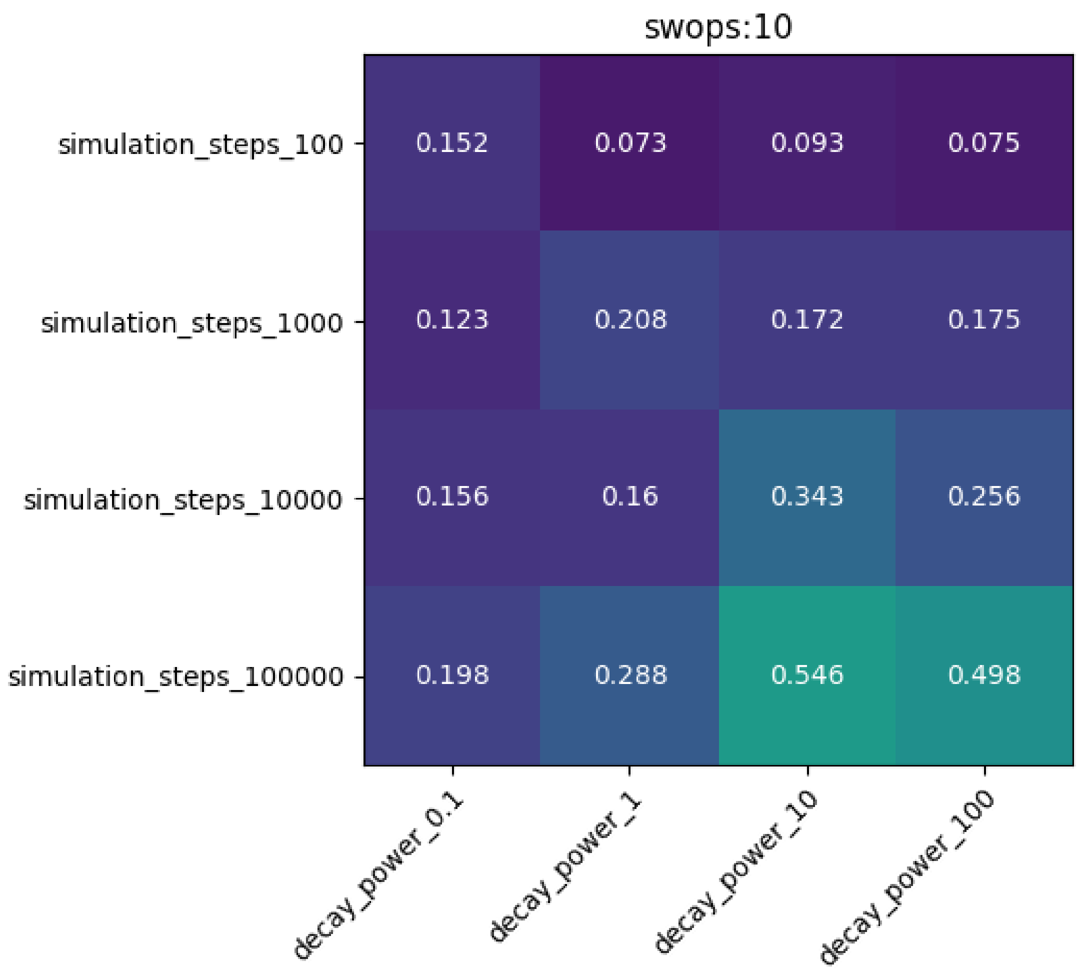
```
swops:10, decay_power:100, steps: 1000 | best_v_final_score: 175, 175
swops:10, decay_power:0.1, steps: 10000 | best_v_final_score: 156, 61
swops:10, decay_power:1, steps: 10000 | best_v_final_score: 160, 160
swops:10, decay_power:10, steps: 10000 | best_v_final_score: 343, 343
swops:10, decay_power:100, steps: 10000 | best_v_final_score: 256, 256
swops:10, decay_power:0.1, steps: 100000 | best_v_final_score: 198, 92
swops:10, decay_power:1, steps: 100000 | best_v_final_score: 288, 288
swops:10, decay_power:10, steps: 100000 | best_v_final_score: 546, 546
swops:10, decay_power:100, steps: 100000 | best_v_final_score: 498, 498
swops:100, decay_power:0.1, steps: 100 | best_v_final_score: 121, 10
swops:100, decay_power:1, steps: 100 | best_v_final_score: 162, 118
swops:100, decay_power:10, steps: 100 | best_v_final_score: 115, 115
swops:100, decay_power:100, steps: 100 | best_v_final_score: 169, 169
swops:100, decay_power:0.1, steps: 1000 | best_v_final_score: 123, 6
swops:100, decay_power:1, steps: 1000 | best_v_final_score: 129, 106
swops:100, decay_power:10, steps: 1000 | best_v_final_score: 202, 202
swops:100, decay_power:100, steps: 1000 | best_v_final_score: 205, 205
swops:100, decay_power:0.1, steps: 10000 | best_v_final_score: 155, 82
swops:100, decay_power:1, steps: 10000 | best_v_final_score: 185, 182
swops:100, decay_power:10, steps: 10000 | best_v_final_score: 237, 237
swops:100, decay_power:100, steps: 10000 | best_v_final_score: 285, 285
swops:100, decay_power:0.1, steps: 100000 | best_v_final_score: 180, 82
swops:100, decay_power:1, steps: 100000 | best_v_final_score: 294, 294
swops:100, decay_power:10, steps: 100000 | best_v_final_score: 360, 358
swops:100, decay_power:100, steps: 100000 | best_v_final_score: 376, 376
```

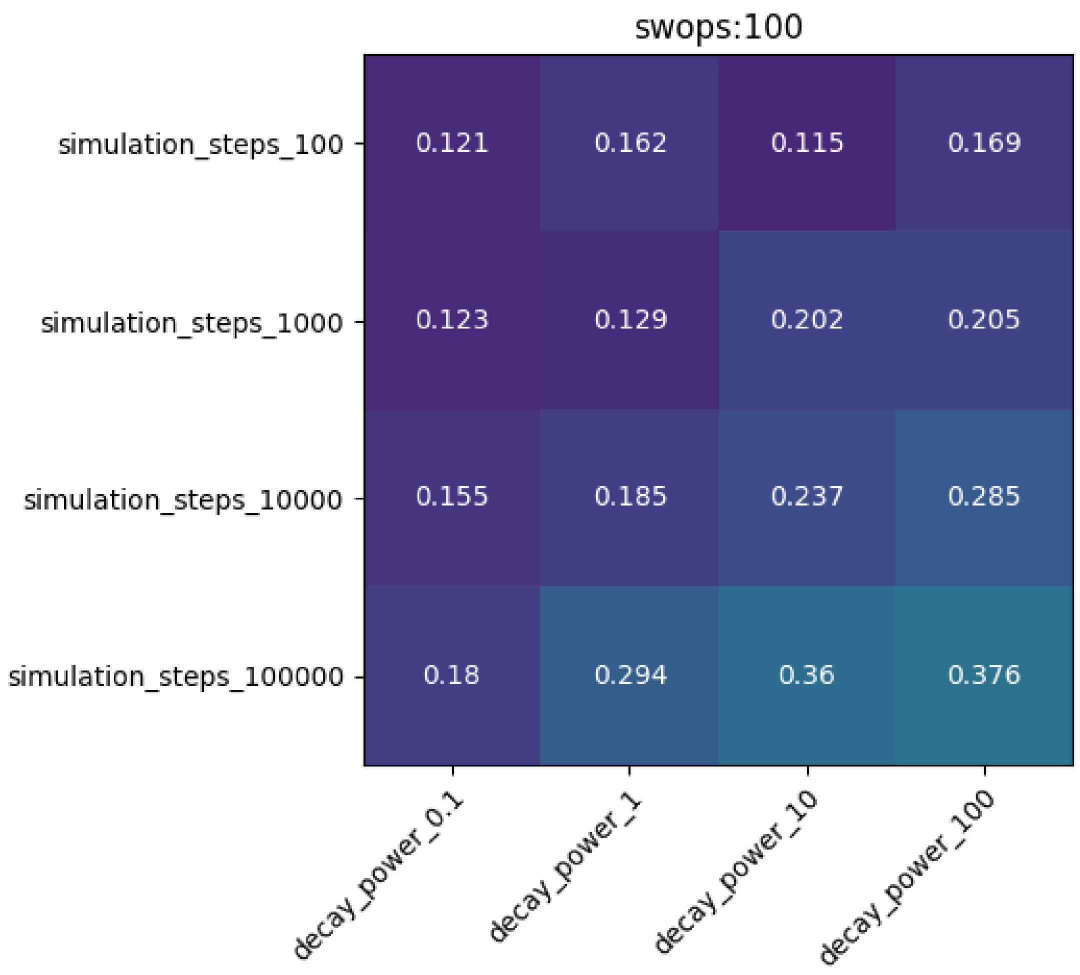
5.2 Generating Heatmaps

Helper functions to help us evaluate the overall relationships between Hyperparameters and Final Scores

[11]:
```python
# Helper Function

def plot_heatmap(experiments: dict, x_labels, y_labels, val_swop: int = 1,␣
 ↪remap_ceiling: int = 1000, save = False, show = True, save_prefix = ''):

    # Construct matrix
    plot_matrix = []
    val_swop = val_swop
    for val_step in step_range:
        plot_matrix.append([])
        for val_decay in decay_power:
            key = str(f'swop{val_swop}_decay{val_decay}_steps{val_step}')
            plot_matrix[-1].append(max(experiments.get(key).get('scor'))) #get␣
 ↪the best score

    # normalize data to remap ceiling
    for row in range(len(plot_matrix)):
        plot_matrix[row] = list(map(lambda x: x/remap_ceiling,␣
↪plot_matrix[row]))

    # initialize plots
    fig, ax = plt.subplots()
    plt.imshow(plot_matrix, vmin = 0, vmax = 1)
    # plt.imshow(plot_matrix, interpolation='nearest')

    x_labels = [str(f'decay_power_{x}') for x in x_labels] # decay
    y_labels = [str(f'simulation_steps_{y}') for y in y_labels] # steps

    # Show all ticks and label them with resp list entries
    ax.set_xticks(range(len(x_labels)), labels = x_labels)
    ax.set_yticks(range(len(y_labels)), labels = y_labels)

    # Rotate the tick labels and set their alignment
    plt.setp(ax.get_xticklabels(),
             rotation = 45,
             ha='right',
             rotation_mode = 'anchor')

    #Loop over data dimensions and create text annotations
    for i in range(len(y_labels)):
        for j in range(len(x_labels)):
            ax.text(j, i, plot_matrix[i][j], ha='center', va='center', color =␣
↪'white')

    # Set titles
    title = str(f'swops:{val_swop}')
    ax.set_title(title)
    save_name = str(f'[{save_prefix}] Heatmap-swop{val_swop}')

    if save:
        plt.savefig('img/' + save_name)

    if show:
        plt.show()

    plt.close()

# Generate heatmaps
for num in num_swops:
    plot_heatmap(experiment_log, x_labels = decay_power, y_labels = step_range,␣
↪val_swop = num, remap_ceiling=1000, show = True, save = True,␣
↪save_prefix=save_prefix)
```


swops:1
simulation_steps_100
simulation_steps_1000
simulation_steps_10000
simulation_steps_100000
decay_power_0.1
decay_power_1
decay_power_10
decay_power_100
0.072
0.073
0.072
0.073
0.07
0.09
0.087
0.177
0.134
0.135
0.354
0.283
0.192
0.376
0.913
0.861


swops:10
simulation_steps_100
simulation_steps_1000
simulation_steps_10000
simulation_steps_100000
decay_power_0.1
decay_power_1
decay_power_10
decay_power_100
0.152
0.073
0.093
0.075
0.123
0.208
0.172
0.175
0.156
0.16
0.343
0.256
0.198
0.288
0.546
0.498



---

6 Report on Experiment

6.1 A Discrete & Logarithmic Experimental Space

We systematically explored the combinations between 3 hyperparameters:

1. Number of Pairswops: *how many neighbours were swopped at each expansion step?*
2. Number of Simulation Steps: *how many steps would we take in the simulation?*
3. Decay Function for Temperature: *what is the decay function for the temperature?*

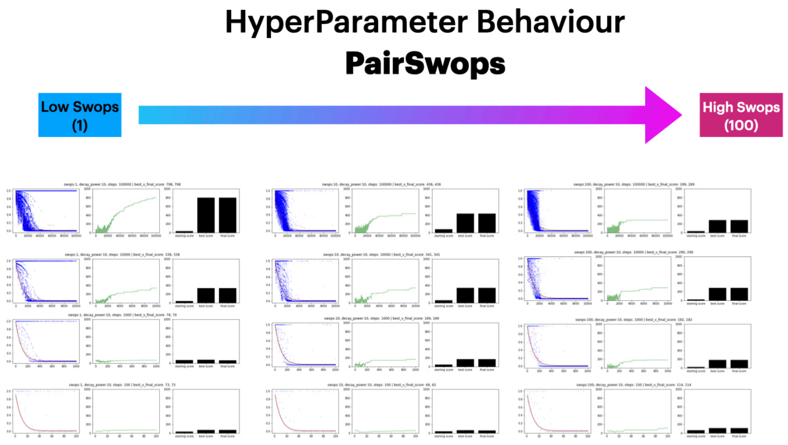
Given the overall search space is incredibly huge, we decided to work with a discrete logarithmic scale of increments for our hyperparameters to explore a large enough portion of this search space. Here is an extract of the ranges explored:

- num_swops = [1, 10, 100]

- step_range = [100, 1000, 10000, 100000]
- decay_power = [0.1, 1, 10, 100]

6.2 Observations on Hyperparameter Behaviours

See Figure Explanation for axes description.

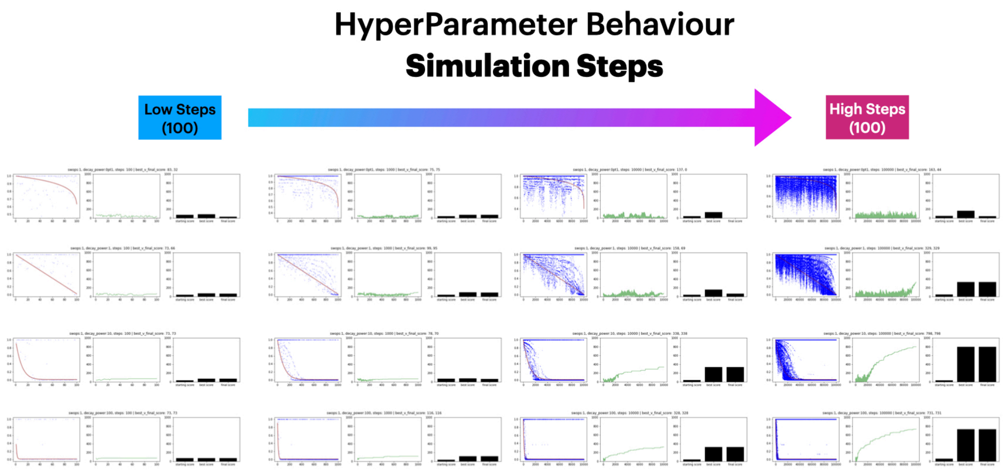


Pairswop Behaviour

Within our experimental numbers, the number of Pairswops are inversely proportional to the score ceiling.

(1) All Pairswops of N=100 did worse than Pairswops of N=1 or N=10. We believe this is because when the Pairswap amounts to a critical percentage of the total graph vertices, vertices already in the correct sequence run a higher risk of getting switched out during neighbour generation and then getting accepted by the annealing process.

(2) Some Pairswops of N=10 outperformed Pairswops of N=1. This has been observed to occur when the simulation steps were increased from 10 to 100. We believe that at such low simulation steps where convergence cannot take place, having higher Pairswops simply increases the chances for more vertices to be randomly placed into the correct sequence.



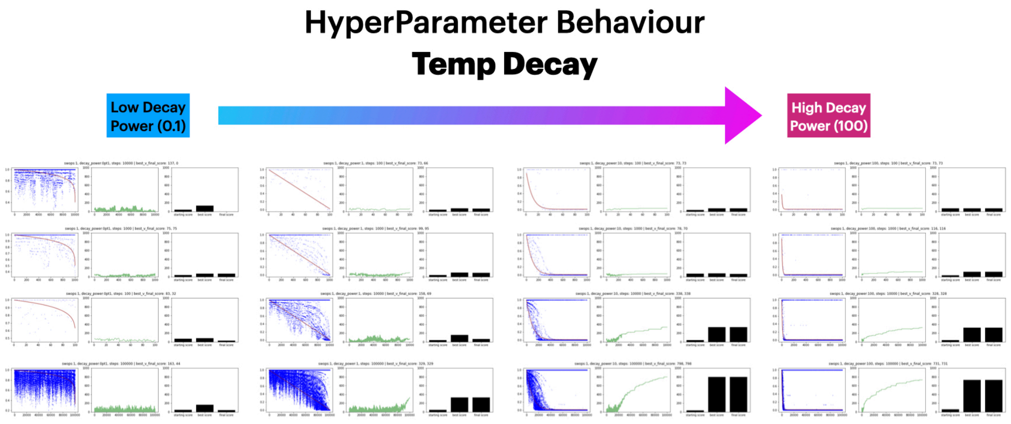
No of Simulation Step Behaviour

Within our experimental numbers, the simulation steps are the single biggest determinant of the score ceiling. The greater the simulation steps, the greater the score ceiling.

(1) The max overlap score always increases when the number of simulation steps increases. This is expected given that Simulated Annealing is a stochastic algorithmn and running the random neighbour generation for an extended period of time yields better chances of getting more sequences right.

(2) DecayPower < 1 is observed to generally have the worst performance, especially at high step size. This is in line with our understanding of Simulated Annealing algorithm - the low rate of decay allows bad neighbour states to continue to be accepted late into the iteration stages, undoing all the progress made.

(3) There is a significant boost in score from the 10^4 to 10^5 step size. The overlap score lifts significantly when there is a long tail in the decay for the temperature. This can generally be attributed to continued prolonged neighbour generation but only the best performing neighbours are now being accepted.



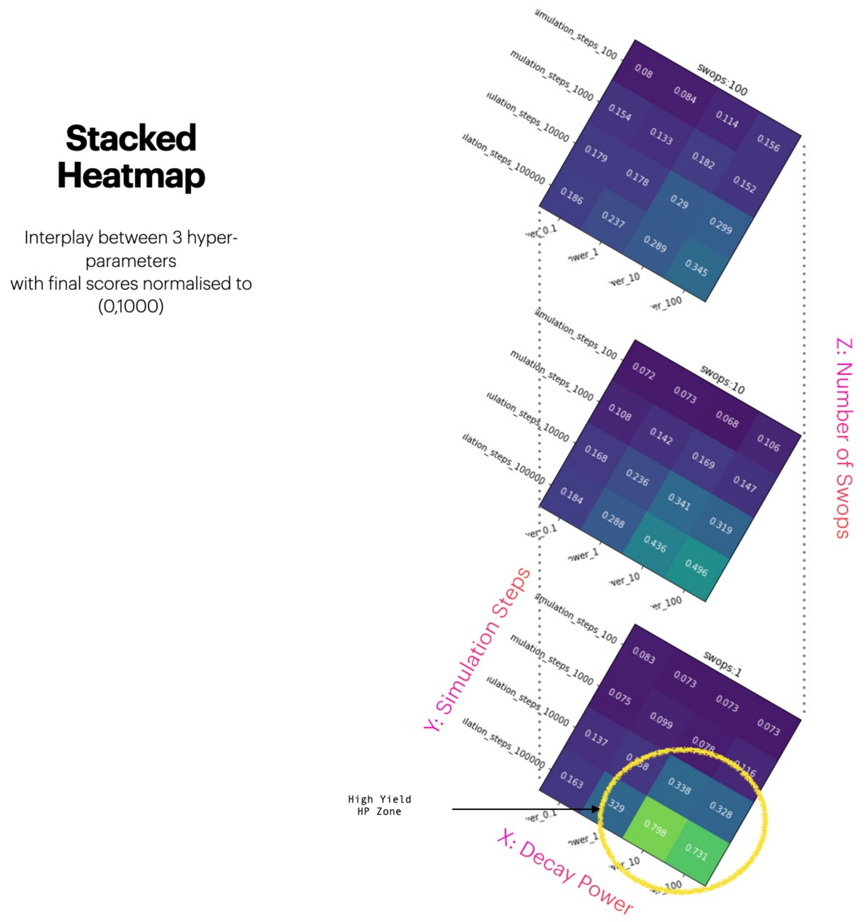
Temperature Decay Behaviour

Within our experimental numbers, there is a balance to be found between "exploration" (which creates the greatest gains) and "convergence" (which is slow and incremental) when it comes to the temperature decay function.

(1) Simulated Annealing algorithm performs better when there is a long tail in the temperature decay function. From our experiments, we observe that DecayPower = 10 has a good mix of exploration (noise in the overlap scores) and convergence (eventual steady state and lift in the score).

(2) For DecayPower < 1, the final score is always worse off than the best score achieved (during the course of the simulation). We believe that this is due to insufficient time for convergence to occur. The relaxed acceptance criteria sends the solution state bouncing between high and low energy levels and creates erratic outcome that is no better than a randomly generated solution.

7 Conclusions to Experiment



Overall chart showing the big picture view of how all the hyperparamaters interact vis a vis final score (range 0-1000 normalised to 0-1)

Macro Conclusions At a macro level, we observe the sweet spot within the experiment parameters are at Pairswop = 1, Decay Power = 10 and simulation steps = 10^5. (see circled region in heatmap).

It is interesting that the best result outperformed some adjacent cells by a factor of 2. (0.329 vs 0.798). Given that we are working with parameter increments in the power of 10, this suggests a logarithmic relationship between each hyperparameter and the final scores - we may see great improvements early on but any further adjustments only bring diminishing returns. We would need to adapt this simulated annealing algorithm with perhaps a neighbour generation function based on heuristics.

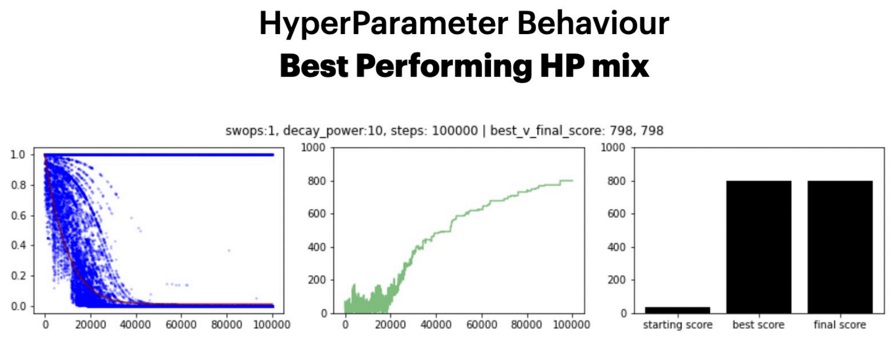


Chart showing the best candidate parameters - a narrative of the best performing experiment

Micro Conclusions At a micro level, the initial exploration is noisy but allows for greater gains. As the temperature decreases, the rate of improvement stabilises, but still increases with a decreasing gradient indicating a convergence on a solution.

8 Thoughts for Further Experiments

8.1 Statistical Trends (Spread and Central Tendency)

Given the stochastic nature of this search algorithm, it would be revealing to explore the central tendency and spread of neighbours generated for each different state.

This will help us appreciate the range of results within a given set of hyperparameters. It would give insight on how many experiments should be done within the same parameter range for a higher confidence level of the best outcome.

Insofar as neighbour generation is concerned, we have been using a binomial distribution function with our random selection. It would be fitting to do a mathematical/empirical analysis of the type of distribution function and their effect on the scores of neighbour states.

We also propose a neighbour generation function based on certain heuristics. Intuitively, we want to keep correctly ordered sequence and instead generate neighbours by swopping vertices not already in order.

8.2 Chained Explorations

It will also be revealing to see what happens if results from one set of hyper-parameters were passed to another set of hyperparameters as an initial state.

We suspect that if a local maxmimum is reached within one set of hyper-parameters, pushing it into another hyperparameter set could yield significant improvements in scores. But careful study needs to done because when initial states are already sub-optimised, there exists a tipping point where relaxed acceptance condition will be counterintuitive and can potentially undo correct sequences.

— end —