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

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

2.0.1 Create Graph Object

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

It follows the format:

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

[161]:
```
import copy
from typing import Type
```

```python
class Graph:
    """
    Taken from AIME4e, search.py
    Modified to include __repr__.

    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 # What is setdefault?

    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)
```

2.0.2 Extract CSV

[162]:
```
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]])
#print (data)

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

"""
Test and Debug
"""
pprint (data_reads[:5])
```

```
[[0,
  'CTTGAATTGGTTCCTGGTATCATTAGGATCTCTGTCTCTCAACATCTGTTTAAGTTCATCGAGAACCACCTCCTCAT
TTTCCAGATAGTCAAACATTTTGACTGAATAGAAGTGAATGAGCTACTGTGAACTCTATACACCCGCACAACTAATGTCA
TTAAATATCATTTTTGAATGTATTTATACCATGTCAAAAACTTGTACAATTATTAATAAAAATAATTAGTGTTTAAATTT
TACCAGTTCCAGATTTTACACCTCCGTTAACACCTCCATTAACCCCACTTTTTACACCACTGGACGATCCTCCTCCCCAC
ATTCCACTGCCACTAGATGTATAAGTTTTAGATCCTTTATTACTACCATCATGTCCATGGATAAAGACACTCCACATGCC
GCCACTACTACCCCCT'],
 [1,
  'ATCTTTAACGAACATATACCTAGATGGTTATTTACTAACAGACATTTTTTCAAGATCTATTGACAATAACTCCTATA
GTTTCCACATCAACCAAGTAATGATCATCTATTGTTATATAACAATAACATAACTCTTTTCCATTTTTATCAGTATCTAT
ATCAACGTCGTTGTAGTGAATAGTAGTCATTGATCTATTATATGAAACGGATATGTCTAGTTAATATTTTCTTTGATTTA
```

```
AAGTCTATAGTCTTTACAAACATAATATCCTTATCCGACTTTATATTTCCTGTAGGGTGGCATAATTTTATTCTGCCTCC
ACAATCAGTGTTTCCAAATATATTACTAGACAATATTCCATATAGT'],
 [2,
  'TTGTACATGTAATGATTTAAAATGTGTAGTCATGCTTATTGATAAAGATCTAAAAATTAAAGCGGGTCCTCGGTACG
TGCTTAACGCTATTAGTCCTCATGCCTATGATGTTTTTAGAAAATCTAATAACTTGAAAGAGATAATAGAAAATGCAGCT
AAACAAAATCTAGACTCTATATCTATTTCTGTTATGACTCCAATTAATCCCATGTTAGCGGAATCATGTGATTCTGTCAA
TAAGGCGTTTAAAAAATTTCCATCAGGAATGTTTGCGGAAGTCAAATACGATGGTGAAAGAGTACAAGTTCATAAAAAAA
ATAACGA'],
 [3,
  'TACTAGATTTTATCTCTAGCGAGATTGTTTAGAATCATTTATCATAACTATGTTTAATAAATTCATCAACGAATATC
GATAAAGACCTCTTGTAATTCGAGTATAGGAAGTAGTATTACCATATCAACTTCCGAGTTAACAATTACTCTAAAACATG
AGGATTGTACTCCTGTCTTTATTGGAGATCACTATTTAGTCGTTGATAAACTAGTAACCTCAGGTTTCTTTACAAACGAT
AAAGTACAACATCAAGACCTCACAACACAGTGCAAGATTAATCTAGAAATCAAATGTAATTCTGGAGGAGA'],
 [4,
  'CTGTCAGAAAATACCATCCAAGCCAATATCTTTACTTTTGTTCATCAGACGTTCCGGAAAGAGGTCCTCAGGTAGGT
TTAGTATCTCAATTGTCTGTCTTGAGTTCCATTACAAATATACTAACGTCTGAGTATTTGGATTTGGAAAAGAAAATTTG
TGAGTATATCAGATCATATTATAAAGATGATATAAGTTACTTTGAAACAGGATTTCCAATCACTATAGAAAATGCTCTAG
TCGCATCTCTTAATCCAAATATGATATGTGATTTTGTAACTGACTTTAGACGTAGAAAACGGATGGGATTCTTCGGTAAC
T']]
```

2.0.3 Generate Directed Graph

A function called overlap is created to generate a graph.

[163]:
```
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)


"""
DEBUG AND TESTING
"""
for node in LOOKUP_GRAPH.nodes()[:10]:
    print (f'{node} : {LOOKUP_GRAPH.get(node)}')
```

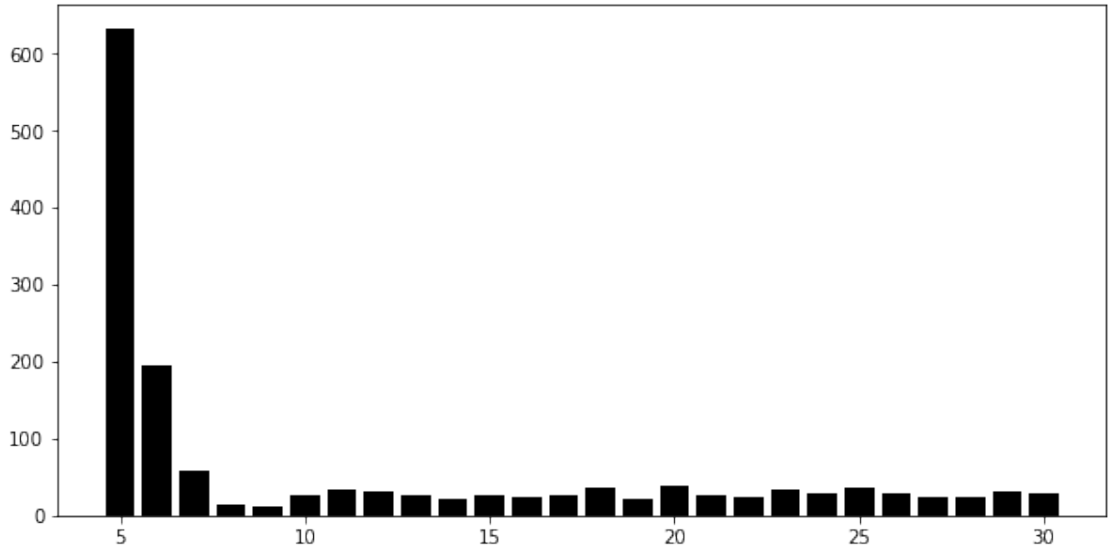
```
0 : {224: 16, 302: 5}
1 : {228: 18, 344: 6, 511: 5}
2 : {223: 22}
3 : {355: 24, 548: 5}
4 : {436: 17}
5 : {185: 15, 102: 5, 156: 5}
6 : {433: 12, 22: 7, 485: 7}
7 : {231: 25}
8 : {590: 23, 256: 6}
9 : {43: 20}
```

2.1 Plotting

Simple histogram to see distribution of overlap # and count

[164]:
```
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()

"""
DEBUGGING AND CHECK
"""
# pprint (histogram)
```



[164]: '\nDEBUGGING AND CHECK\n'

3 Part 2 - Implementation of Local Seach using Simulated Annealing

3.1 Problem Class

[165]:
```
class Problem(object):
    """The abstract class for a search problem."""

    def __init__(self, initial=None, goals=(), **additional_keywords): # can be␣
 ↪refactored to be shorter
        """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
```

3.2 TSP Subclass

[166]:
```
# 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)


"""
Debug and Testing
"""

tsp_problem = TSP(LOOKUP_GRAPH.nodes(), lookup_graph = LOOKUP_GRAPH, swops=500)
print (tsp_problem.path_cost(tsp_problem.initial))
print (tsp_problem.N_opt(tsp_problem.initial))
```

```
43
[189, 190, 191, 314, 42, 490, 475, 398, 35, 210, 156, 548, 34, 33, 298, 5, 4,
77, 555, 193, 9, 378, 106, 105, 89, 557, 535, 258, 257, 246, 576, 2, 3, 196, 74,
161, 491, 472, 135, 45, 46, 533, 433, 151, 524, 523, 165, 536, 561, 93, 547,
131, 446, 182, 44, 43, 78, 496, 373, 152, 251, 254, 545, 544, 543, 326, 327, 67,
```

```
166, 530, 158, 451, 450, 286, 230, 275, 274, 124, 141, 140, 322, 180, 260, 580,
367, 442, 478, 256, 247, 449, 408, 214, 406, 301, 469, 468, 467, 345, 218, 392,
300, 425, 424, 264, 265, 595, 596, 163, 187, 308, 509, 409, 470, 379, 118, 127,
412, 411, 410, 532, 542, 526, 527, 420, 293, 117, 29, 55, 443, 28, 70, 455, 456,
457, 279, 232, 233, 128, 192, 389, 388, 194, 464, 465, 586, 485, 304, 303, 570,
252, 445, 229, 441, 136, 315, 150, 145, 241, 26, 25, 24, 48, 94, 95, 115, 114,
363, 558, 181, 495, 559, 489, 348, 347, 339, 267, 266, 551, 242, 447, 12, 434,
197, 132, 567, 566, 487, 271, 66, 454, 494, 381, 380, 550, 53, 263, 510, 17, 16,
15, 323, 575, 239, 325, 334, 333, 21, 396, 13, 39, 79, 512, 497, 200, 569, 432,
431, 438, 107, 462, 493, 317, 463, 138, 84, 429, 428, 427, 541, 540, 413, 414,
312, 335, 452, 157, 402, 394, 393, 337, 253, 585, 474, 473, 531, 336, 1, 126,
482, 556, 86, 85, 476, 453, 437, 183, 184, 146, 97, 96, 237, 320, 255, 546, 130,
137, 268, 80, 201, 511, 504, 503, 58, 488, 6, 343, 23, 289, 288, 310, 215, 284,
321, 477, 386, 385, 507, 435, 330, 176, 177, 513, 31, 30, 243, 129, 205, 342,
508, 357, 422, 358, 32, 371, 372, 198, 261, 164, 554, 61, 60, 220, 221, 534, 63,
592, 593, 594, 584, 340, 341, 91, 92, 444, 162, 213, 564, 563, 560, 188, 186,
306, 305, 225, 573, 502, 331, 309, 351, 234, 316, 597, 399, 98, 147, 154, 120,
119, 38, 359, 360, 102, 168, 236, 583, 217, 216, 350, 591, 537, 538, 572, 59,
206, 207, 208, 209, 36, 426, 458, 228, 401, 492, 219, 361, 103, 54, 395, 160,
159, 299, 276, 212, 483, 291, 56, 22, 307, 170, 419, 273, 123, 122, 590, 521,
522, 423, 332, 14, 202, 370, 459, 460, 461, 565, 552, 280, 382, 553, 172, 231,
287, 75, 76, 369, 562, 539, 324, 439, 224, 223, 204, 589, 391, 47, 397, 283,
175, 68, 104, 471, 195, 377, 376, 375, 374, 440, 404, 403, 549, 415, 400, 41,
417, 418, 272, 352, 64, 65, 8, 69, 244, 245, 110, 109, 269, 390, 7, 112, 277,
515, 281, 179, 178, 514, 171, 259, 49, 81, 318, 353, 83, 486, 416, 144, 262,
383, 99, 292, 285, 498, 499, 500, 501, 407, 153, 87, 240, 574, 27, 368, 222, 18,
19, 62, 149, 148, 142, 88, 73, 516, 517, 518, 250, 235, 270, 313, 311, 346, 338,
297, 356, 295, 116, 479, 211, 506, 155, 529, 203, 587, 238, 133, 484, 139, 185,
578, 579, 329, 10, 384, 100, 101, 167, 387, 430, 296, 571, 505, 481, 480, 294,
111, 278, 525, 37, 169, 82, 57, 143, 50, 51, 249, 248, 173, 125, 174, 355, 354,
421, 362, 577, 405, 52, 290, 349, 520, 519, 11, 466, 344, 328, 90, 568, 588,
319, 134, 72, 71, 581, 366, 365, 364, 113, 121, 528, 108, 582, 436, 448, 302,
199, 40, 226, 227, 282, 0, 20, 598]
```

3.3 Node Class

Nodeclass from AIMA. Extensions to allow expansion and path.

[167]:
```
# 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)) # potential to log all the temperature, etc here
        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))
```

3.4 Simulated Annealing Definition

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

[168]:
```
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

'''
Debugging & Testing
'''
```

```
#test for Probability.
count_true = 0
prob = 0.25
for i in range(100):
    count_true += probability (prob)
print (f'Input probability is {prob}. Out of 100 tries, there are {count_true}␣
 ↪"Trues"')

#test for simulated anneal
states = simulated_annealing_full(tsp_problem)
print (type(states))
print (states.keys())
print (type(tsp_problem))

#exploring the tsp problem  output
print (tsp_problem)
print (states.keys())
print (len(states.get('state')))
```

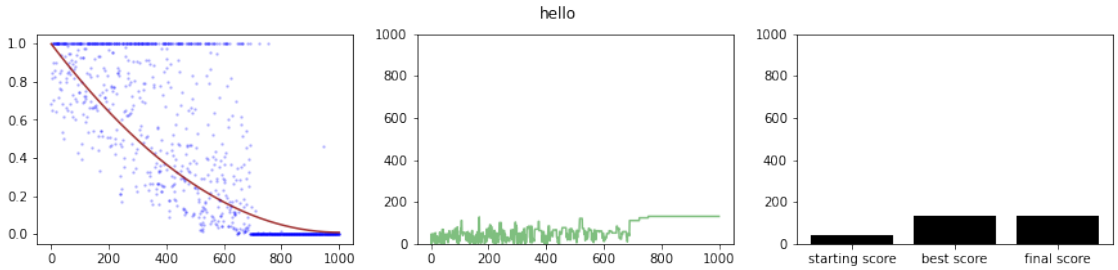
```
Input probability is 0.25. Out of 100 tries, there are 14 "Trues"
<class 'dict'>
dict_keys(['temp', 'accp', 'delt', 'step', 'scor', 'prob', 'state'])
<class '__main__.TSP'>
<__main__.TSP object at 0x7fb1fde658b0>
dict_keys(['temp', 'accp', 'delt', 'step', 'scor', 'prob', 'state'])
1000
```

3.5 Graphing Function

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

[169]:
```
# 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 = 'hello', show = True)
```



4 Exercise

4.1 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 'next' function.

[170]:
```
"""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: 83, 32
swops:1, decay_power:1, steps: 100 | best_v_final_score: 73, 66
```

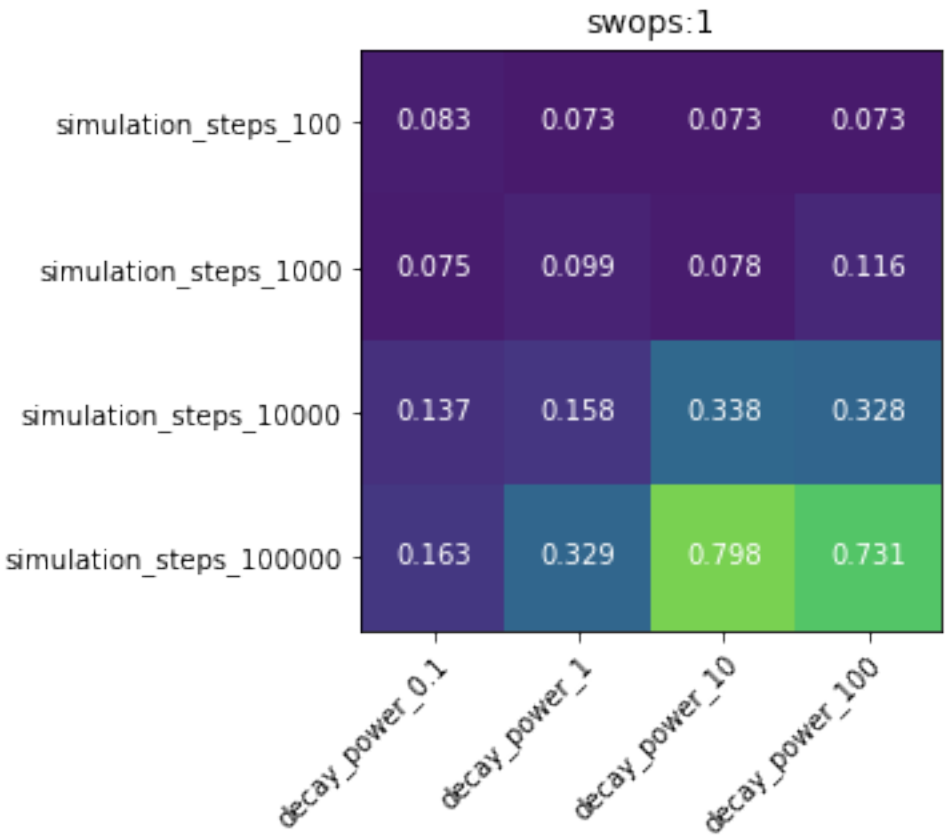
swops:1, decay_power:10, steps: 100 | best_v_final_score: 73, 73
swops:1, decay_power:100, steps: 100 | best_v_final_score: 73, 73
swops:1, decay_power:0.1, steps: 1000 | best_v_final_score: 75, 75
swops:1, decay_power:1, steps: 1000 | best_v_final_score: 99, 95
swops:1, decay_power:10, steps: 1000 | best_v_final_score: 78, 70
swops:1, decay_power:100, steps: 1000 | best_v_final_score: 116, 116
swops:1, decay_power:0.1, steps: 10000 | best_v_final_score: 137, 0
swops:1, decay_power:1, steps: 10000 | best_v_final_score: 158, 69
swops:1, decay_power:10, steps: 10000 | best_v_final_score: 338, 338
swops:1, decay_power:100, steps: 10000 | best_v_final_score: 328, 328
swops:1, decay_power:0.1, steps: 100000 | best_v_final_score: 163, 44
swops:1, decay_power:1, steps: 100000 | best_v_final_score: 329, 329
swops:1, decay_power:10, steps: 100000 | best_v_final_score: 798, 798
swops:1, decay_power:100, steps: 100000 | best_v_final_score: 731, 731
swops:10, decay_power:0.1, steps: 100 | best_v_final_score: 72, 6
swops:10, decay_power:1, steps: 100 | best_v_final_score: 73, 70
swops:10, decay_power:10, steps: 100 | best_v_final_score: 68, 62
swops:10, decay_power:100, steps: 100 | best_v_final_score: 106, 106
swops:10, decay_power:0.1, steps: 1000 | best_v_final_score: 108, 20
swops:10, decay_power:1, steps: 1000 | best_v_final_score: 142, 142
swops:10, decay_power:10, steps: 1000 | best_v_final_score: 169, 169
swops:10, decay_power:100, steps: 1000 | best_v_final_score: 147, 147
swops:10, decay_power:0.1, steps: 10000 | best_v_final_score: 168, 35
swops:10, decay_power:1, steps: 10000 | best_v_final_score: 236, 236
swops:10, decay_power:10, steps: 10000 | best_v_final_score: 341, 341
swops:10, decay_power:100, steps: 10000 | best_v_final_score: 319, 319
swops:10, decay_power:0.1, steps: 100000 | best_v_final_score: 184, 104
swops:10, decay_power:1, steps: 100000 | best_v_final_score: 288, 288
swops:10, decay_power:10, steps: 100000 | best_v_final_score: 436, 436
swops:10, decay_power:100, steps: 100000 | best_v_final_score: 496, 496
swops:100, decay_power:0.1, steps: 100 | best_v_final_score: 80, 23
swops:100, decay_power:1, steps: 100 | best_v_final_score: 84, 76
swops:100, decay_power:10, steps: 100 | best_v_final_score: 114, 114
swops:100, decay_power:100, steps: 100 | best_v_final_score: 156, 156
swops:100, decay_power:0.1, steps: 1000 | best_v_final_score: 154, 31
swops:100, decay_power:1, steps: 1000 | best_v_final_score: 133, 130
swops:100, decay_power:10, steps: 1000 | best_v_final_score: 182, 182
swops:100, decay_power:100, steps: 1000 | best_v_final_score: 152, 152
swops:100, decay_power:0.1, steps: 10000 | best_v_final_score: 179, 121
swops:100, decay_power:1, steps: 10000 | best_v_final_score: 178, 174
swops:100, decay_power:10, steps: 10000 | best_v_final_score: 290, 290
swops:100, decay_power:100, steps: 10000 | best_v_final_score: 299, 299
swops:100, decay_power:0.1, steps: 100000 | best_v_final_score: 186, 48
swops:100, decay_power:1, steps: 100000 | best_v_final_score: 237, 230
swops:100, decay_power:10, steps: 100000 | best_v_final_score: 289, 289
swops:100, decay_power:100, steps: 100000 | best_v_final_score: 345, 345

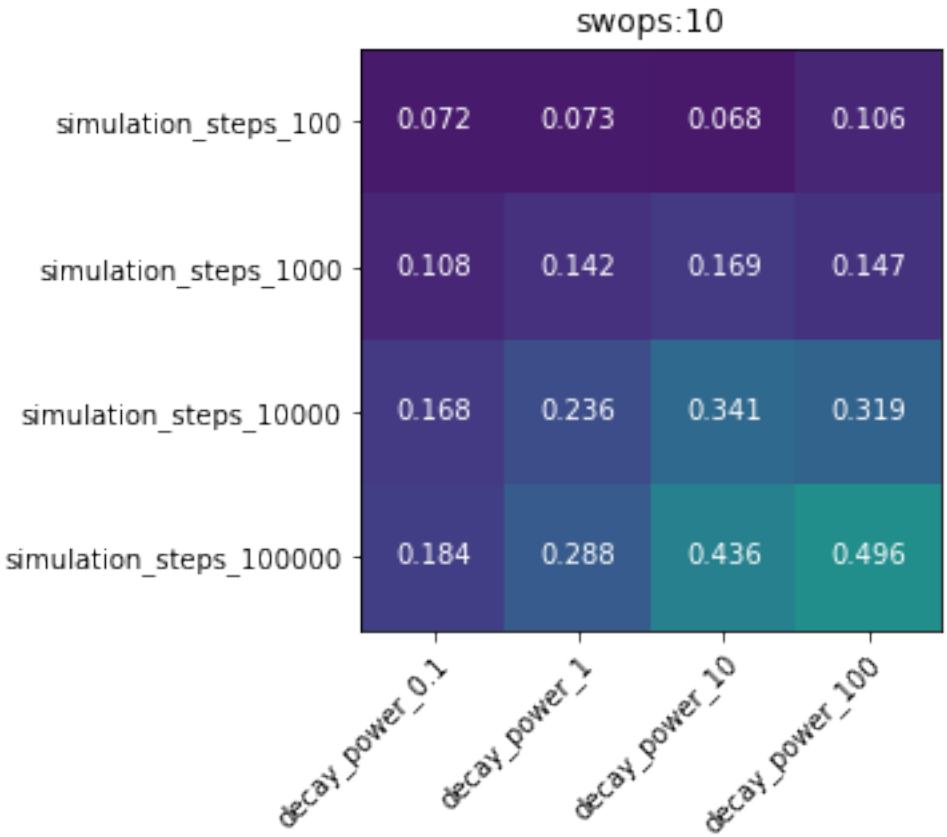
4.2 Generating Heatmaps

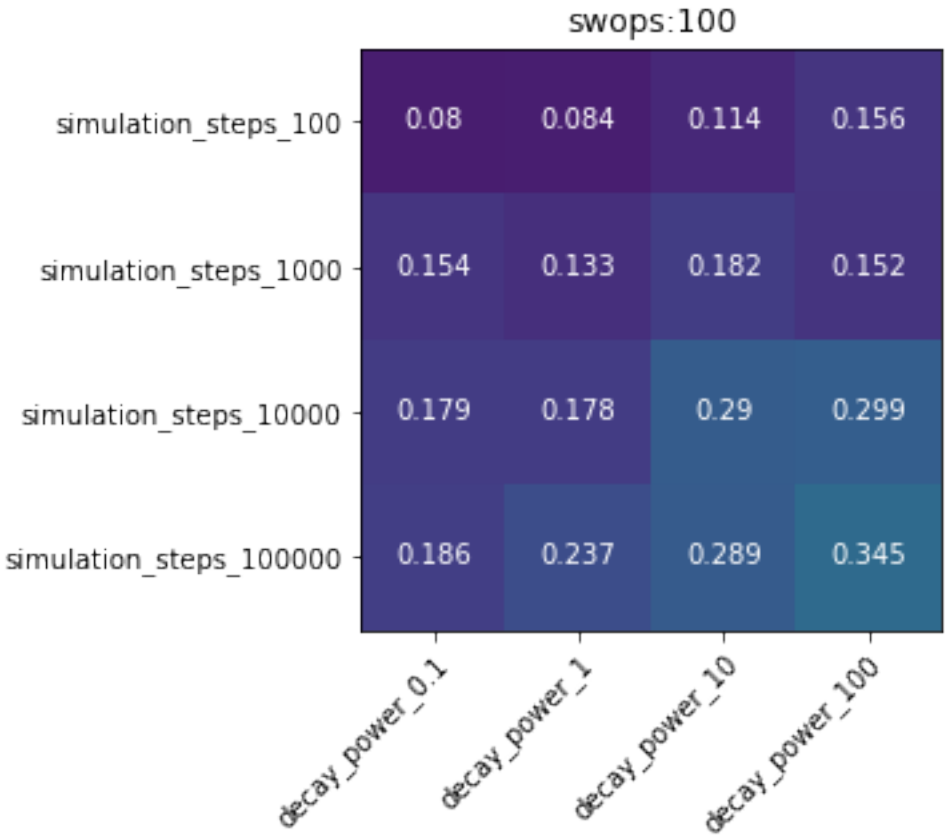
Helper functions to help us evaluate the overall relationships between HPs and Final Scores

[172]:
```
# 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:10
simulation_steps_100
simulation_steps_1000
simulation_steps_10000
simulation_steps_100000
0.072
0.073
0.068
0.106
0.108
0.142
0.169
0.147
0.168
0.236
0.341
0.319
0.184
0.288
0.436
0.496
decay_power_0.1
decay_power_1
decay_power_10
decay_power_100



---

5 Report on Experiment

5.1 Experimental Space

We systematically explored 3 hyperparameters:

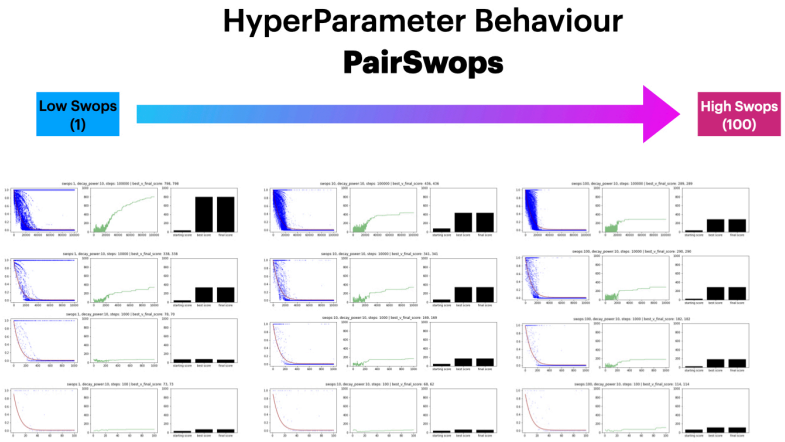
(1) **Number of Pairswops**: how many neighbours were swopped at each expansion step

(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 inifinite, we decided to work in a logarithmic scale of increments to explore a larger subspace for HPs.

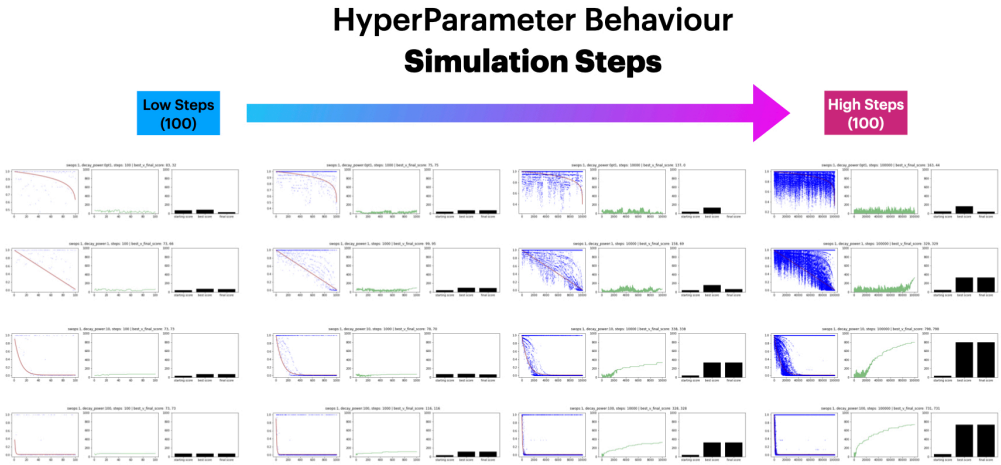
5.2 Hyperparameter Behaviours



Within the 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 takes a larger % of the total graph nodes, the chances of a better score get worse.

(2) Some pariswops of N=10 outperformed pairswops from N=1. This occured when the simulation steps were increased from 10 to 100.



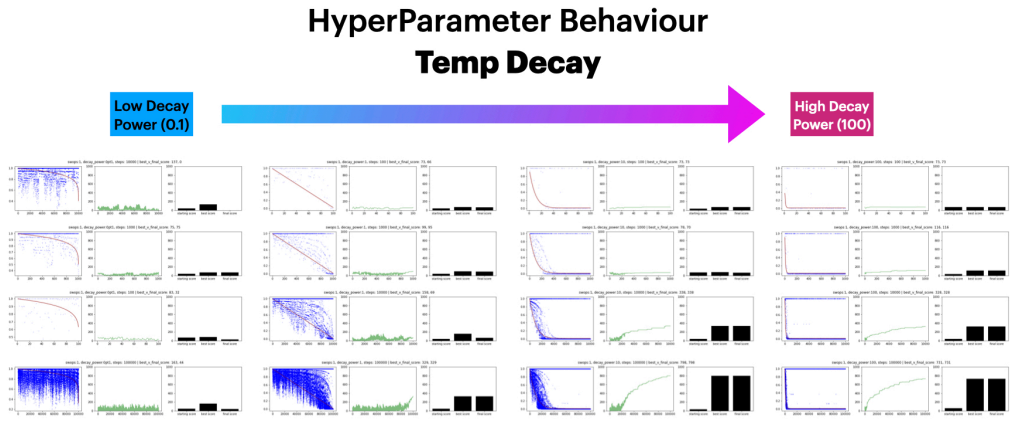
Within the experimental numbers, the simulation steps are the single biggest determinant of the score ceiling. The greater the simulation steps, the greater the score ceiling.

(1) The max score achieved always increased when the number of simulation steps increased.

(2) The best score performed worse for DecayPower = -1. We believe this is because the low rate of decay with high number of steps allowed the SA process to go off track.

(3) For all other best scores, they improved with increasing simulation steps. What is interesting is that there was a significant boost from 10000 to 10000. The scores lift significantly when there is a long tail in the decay for the

It is also interesting to note that the general shape of the score curve remains similar for the same number of steps. e.g. from step 100 to step 10000 for Decay Power of 10, there is very little noise in the beginning and a convex increase in score.



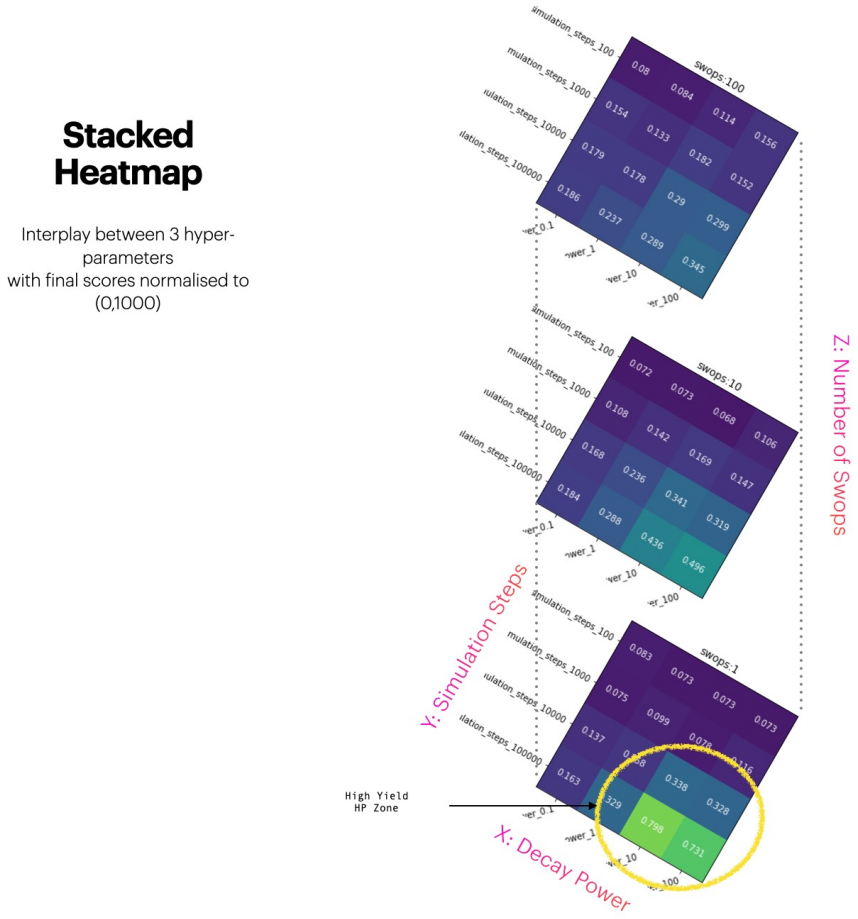
Within the experimental numbers, there is a balance to be found between global exploration (which creates the greatest gains) and local optimization (which is slow and incremental).

(1) The temperature decay performed better when there was a long tail. This was the biggest determinant of how well the best score. We observe that T=10 has a good mix of exploration (noise in the scoring) and optimization (lift in the score).

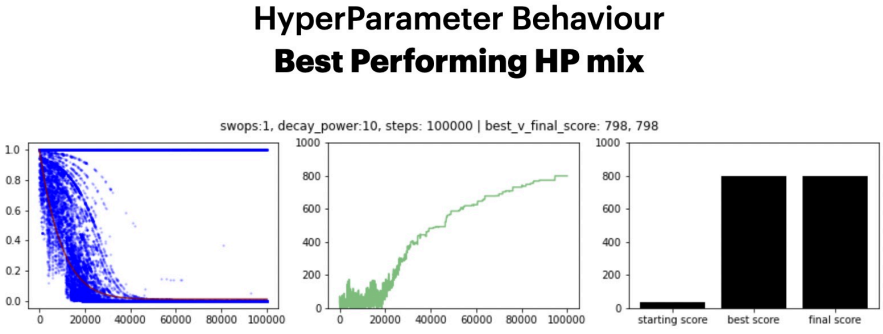
(2) For temperature decay < 0, the final score was always worse off than the best score. We believe that this is due to insufficient time for optimization. The relaxed acceptance criteria sends the processes bounching between high and low energy levels and creates erratic behaviour.

---

6 Conclusions to Experiment



Overall



Best Candidate

The chart above takes a big picture view of how all the hyperparamaters (HPs) interact vis a vis a the final score. It also pulls out the exploration history of the best performing experiment.

Overall, the sweet spot within the experiment parameters are at 1x Pairswop, decay power = 10 and simulation steps = 1000000. (see circled region in heatmap).

It is interesting that it outperformed some adjacent cells by a factor of 2. (0.329 vs 0.798). Given

that we are working with HP increases in the power of 10, this suggests a logarithmic relationship between the HP and the final scores.

We believe at this set of HPs, there is still further gains to be made in score because the gradient of the scorecurve is still on ascent and has not flatlined yet.

---

7 Thoughts for Further Experiments

7.0.1 Statistical Trends: Spread and Central Tendency

Given the stochastic nature of this search algorithm, it would be interesting to explore the central tendency and spread for the different states. This will help us appreciate the range of results within a given set of hyperparameters and how many experiments should be done.

7.0.2 Chained Explorations

It will be revealing to see what happens if results from one set of hyper-parameters were given to another set of hyperparameters. We believe that if a local maxmimum is reach within one set of hyper-parameters, pushing it into another HP set could yield some marginal (or significant) increase in scores.

We anticipate that when initial conditions are more optimized, there will be a tipping point where relaxed search conditions will be counter intuitive to boost the scores.

— end —

[ ]: