Cyberpwned: Beating Cyberpunk 2077's hacking minigame in 50 lines of Python

December 20, 2020

If you have a passing interest in anything gaming related and haven't been living under a rock for the last couple years, you're probably somewhat familiar with Cyberpunk 2077. After a long and repeatedly extended wait, it's finally here! And it has a hacking minigame! And getting a good score gets you better gear! Can some quick Python magic give us the edge we need in this cruel, cruel Net? You bet.

A quick overview of the minigame: You're given a square matrix and one or more sequences of hexadecimal numbers, along with a buffer of certain length. Your goal is to complete the most amount of sequences choosing as many nodes as the buffer allows. Each sequence will advance in progress if the node chosen is the next node in the sequence. When the game starts, you may choose any of the values in the first row of the matrix. After that, the values you can choose will alternate between the Nth column/row each turn, with N being the index of the last picked value. If this horrible description wasn't helpful, click here for a more detailed one, courtesy of RPS.

All valid paths are considered and the final score of each path is calculated depending on how many sequences were completed by traversing it. The path with the best score is then printed to screen. The algorithm itself isn't particularly interesting nor efficient, unfortunately. However, it should be fast enough to solve any of Cyberpunk's minigames in a few seconds, so fast enough! I'm sure there's some low hanging fruit that could speed up execution,

but I thought I'd share what I have so far and update the post as new ideas/suggestions come up.

Let's start off with the game's components:

- Code Matrix: Multidimensional array containing hexadecimal values (nodes), the "gameboard"

```
code_matrix = [
    [0x1c, 0xbd, 0x55, 0xe9, 0x55],
    [0x1c, 0xbd, 0x1c, 0x55, 0xe9],
    [0x55, 0xe9, 0xe9, 0xbd, 0xbd],
    [0x55, 0xff, 0xff, 0x1c, 0x1c],
    [0xff, 0xe9, 0x1c, 0xbd, 0xff]
]
```

- Buffer: Number indicating maximum number of chosen coordinates

```
buffer_size = 7
```

- Coordinate: Specific point in the code matrix
- Path: Ordered collection of unique coordinates which the user accessed

```
# The same coordinate can't be chosen more than once per path
class DuplicatedCoordinateException(Exception):
    pass

# All paths match the following pattern:
# [(0, a), (b, a), (b, c), (d, c), ...]
class Path():
    def __init__(self, coords=[]):
        self.coords = coords

    def __add__(self, other):
        new_coords = self.coords + other.coords
        if any(map(lambda coord: coord in self.coords, other.coords)):
            raise DuplicatedCoordinateException()
        return Path(new_coords)

    def __repr__(self):
        return str(self.coords)
```

- Sequence: Paths that offer varying quality rewards if completed

```
sequences = [
    [0x1c, 0x1c, 0x55],
    [0X55, 0Xff, 0X1c],
    [0xbd, 0xe9, 0xbd, 0x55],
    [0x55, 0x1c, 0xff, 0xbd]
]
```

- Score: Calculation that can be tweaked to prioritize different rewards

```
class SequenceScore():

    def __init__(self, sequence, buffer_size, reward_level=0):
        self.max_progress = len(sequence)
        self.sequence = sequence
        self.score = 0
        self.reward_level = reward_level
        self.buffer_size = buffer_size

    def compute(self, compare):
        if not self.__completed():
            if self.sequence[self.score] == compare:
                self.__increase()
            else:
                self.__decrease()

    # When the head of the sequence matches the targeted node, increase the score by 1
    # If the sequence has been completed, set the score depending on the reward level
    def __increase(self):
        self.buffer_size -= 1
        self.score += 1
        if self.__completed():
            # Can be adjusted to maximize either:
            #   a) highest quality rewards, possibly lesser quantity
            self.score = 10 ** (self.reward_level + 1)
            #   b) highest amount of rewards, possibly lesser quality
            # self.score = 100 * (self.reward_level + 1)

    # When an incorrect value is matched against the current head of the sequence, the score is decreased
by 1 (can't go below 0)
    # If it's not possible to complete the sequence, set the score to a negative value depending on the
reward
    def __decrease(self):
        self.buffer_size -= 1
        if self.score > 0:
            self.score -= 1
        if self.__completed():
```

```
                self.score = -self.reward_level - 1

        # A sequence is considered completed if no further progress is possible or necessary
        def __completed(self):
            return self.score < 0 or self.score >= self.max_progress or self.buffer_size < self.max_progress -
    self.score

    class PathScore():

        def __init__(self, path, sequences, buffer_size):
            self.score = None
            self.path = path
            self.sequence_scores = [SequenceScore(sequence, buffer_size, reward_level) for reward_level,
    sequence in enumerate(sequences)]

        def compute(self): # Returns the sum of the individual sequence scores
            if self.score != None:
                return self.score
            for row, column in self.path.coords:
                for seq_score in self.sequence_scores:
                    seq_score.compute(code_matrix[row][column])
            self.score = sum(map(lambda seq_score: seq_score.score, self.sequence_scores))
            return self.score
```

The path generation is pretty similar to a DFS graph traversal. The only significant difference is that the nodes accessible will vary depending on the current turn and the previous node's coordinates.

```
# Returns all possible paths with size equal to the buffer
def generate_paths(buffer_size):
    completed_paths = []

    # Return next available row/column for specified turn and coordinate.
    # If it's the 1st turn the index is 0 so next_line would return the
    # first row. For the second turn, it would return the nth column, with n
    # being the coordinate's row
    def candidate_coords(turn=0, coordinate=(0,0)):
        if turn % 2 == 0:
            return [(coordinate[0], column) for column in range(len(code_matrix))]
        else:
            return [(row, coordinate[1]) for row in range(len(code_matrix))]

    # The stack contains all possible paths currently being traversed.
    def _walk_paths(buffer_size, completed_paths, partial_paths_stack = [Path()], turn = 0, candidates =
candidate_coords()):

        path = partial_paths_stack.pop()

        for coord in candidates:
            try:
```

```
                new_path = path + Path([coord])

            # Skip coordinate if it has already been visited
            except DuplicatedCoordinateException:
                continue

            # Full path is added to the final return list and removed from the partial paths stack
            if len(new_path.coords) == buffer_size:
                completed_paths.append(new_path)
            else: # Add new, lengthier partial path back into the stack
                partial_paths_stack.append(new_path)
                _walk_paths(buffer_size, completed_paths, partial_paths_stack, turn + 1, candidate_coords(turn
+ 1, coord))

    _walk_paths(buffer_size, completed_paths)
    return completed_paths
```

Let's run it and check that everything is working correctly. While most of the breach protocol minigames have randomized values, the "Spellbound" quest involves a fixed state and the best solution is already known. Courtesy of GamerJournalist:


```
paths = [(path, PathScore(path, sequences, buffer_size).compute()) for path in generate_paths(buffer_size)]
max_path = max(paths, key=lambda path: path[1])
# [(0, 1), (2, 1), (2, 3), (1, 3), (1, 0), (4, 0), (4, 3)] -> Should traverse 3rd and 4th sequences, finishes
in ~3 seconds
```

```
print(max_path[0])
```

Success! Out of curiosity, I thought I'd try increasing the buffer size until I could find a path which cleared all sequences. Be warned though, the result might take some time to compute since it has to go through every possible path of length 11.

```
buffer_size = 11
paths = [(path, PathScore(path, sequences, buffer_size).compute()) for path in generate_paths(buffer_size)]
max_path = max(paths, key=lambda path: path[1])

# [(0, 0), (1, 0), (1, 3), (3, 3), (3, 1), (0, 1), (0, 3), (2, 3), (2, 0), (4, 0), (4, 2)] -> Should traverse
all sequences, finishes in ~40 seconds
print(max_path[0], max_path[1])
```

OK, optimization time! We'll assume that there's at least 1 path with a non-negative score. That way, we can actually stop traversing a partial path if its current score (adjusted depending on buffer's size and current turn) is lower than 0.

```
def generate_paths(...):
    ...
    def _walk_paths(...):
    ...

                # Full path is added to the final return list and removed from the partial paths stack
                if len(new_path.coords) == buffer_size:
                    completed_paths.append(new_path)
                # If the partial path score is already lower than 0 we should be able to safely stop traversing
it
                elif PathScore(new_path, sequences, buffer_size).compute() < 0:
                    continue
                else: # Add new, lengthier partial path back into the stack
                    partial_paths_stack.append(new_path)
                    _walk_paths(buffer_size, completed_paths, partial_paths_stack, turn + 1,
candidate_coords(turn + 1, coord))
    ...
```

Let's try to get our original result with a buffer of length 7 again:

```
buffer_size = 7
paths = [(path, PathScore(path, sequences, buffer_size).compute()) for path in generate_paths(buffer_size)]
max_path = max(paths, key=lambda path: path[1])
# [(0, 1), (2, 1), (2, 3), (1, 3), (1, 0), (4, 0), (4, 3)] -> Should traverse 3rd and 4th sequences, finishes
instantly
print(max_path[0])
```

It works! The new solution is noticeably faster, but a buffer of length 7 was already calculated pretty quickly, so it's hard to get a sense of the speedup. We'll see how the optimization fares when calculating the most rewarding path possible.

```
buffer_size = 7
paths = [(path, PathScore(path, sequences, buffer_size).compute()) for path in generate_paths(buffer_size)]
max_path = max(paths, key=lambda path: path[1])
# [(0, 0), (1, 0), (1, 3), (3, 3), (3, 1), (0, 1), (0, 3), (2, 3), (2, 0), (4, 0), (4, 2)] -> Should traverse
all sequences, finishes in ~13 seconds
print(max_path[0])
```

A ~3x speedup by adding 2 lines of code. Not bad, if I do say so myself. There's probably more low hanging fruit for optimization opportunities, so I might be updating the post over the week seeing what else I can find. Anyways, that's pretty much it. Feel free to send some suggestions to improve code quality/performance!