Chapter 1

Binary Search

As mentioned in the last chapter, you are going to make a priority queue for use with Dijkstra's Algorithm. Using it will look like this:

```
import kpqueue

myqueue = pqueue.PriorityQueue()
myqueue.add(long_beach, 0) # Inserts first city and its cost
myqueue.add(san_diego, 14) # Puts San Diego after Long Beach
myqueue.add(los_angeles,12) # Inserts LA between Long Beach and San Diego
current_city = myqueue.pop() # Returns first city (Long Beach) and removes it
```

Now if an item gets a new priority, we need to remove it and reinsert it in the new spot.

```
myqueue.add(city_a, 16) # Puts it last in the queue
myqueue.update(city_a, 16, 13) # Moves it to between LA and San Diego
```

1.1 A Naive Implementation of the Priority Queue

Create a file called `kpqueue.py`. Let's do a simple implementation that stores the priority and the data as tuple. And we will keep it sorted by the priority. If two tuples have the same priority, we'll sort by the data.

Type this in to `kpqueue.py`:

```
class PriorityQueue:
    def __init__(self):
        self.list = []

    # Return and remove the first item
    def pop(self):
        if len(self.list) > 0:
            return self.list.pop(0)
        else:
            return None

    def __len__(self):
        return len(self.list)

    def update(self, value, old_priority, new_priority):
        old_pair = (old_priority, value)
        self.list.remove(old_pair)
        self.add(value, new_priority)

    def add(self, value, priority):
        pair = (priority, value)
        # Add it at the end
        self.list.append(pair)
        # Resort the list
        self.list.sort()
```

This will work fine, but it could be much more efficient:

- Every time we add a single element, we resort the whole list.
- The function `remove` is searching the list sequentially for the item to delete.

In a minute, we will revisit these inefficiencies and make the better.

1.2 Using the Priority Queue

We are going to change `graph.py` to use the priority queue. While we are doing, why don't we also shrink the memory footprint of our program a bit.

Notice that as the algorithm is running, each node is in one of three states:

- Unseen: In the earlier implementation, these were the nodes with `math.inf` as their cost.
- Seen, but not finalized: These are "unvisited" but don't have `math.inf` as their cost.
- Finalized: These are the "visited" nodes – we know that their cost won't decrease any more.

We can shrink the memory foot print by not putting the unseen into the dist dictionary at all. And instead of a separate set for "unvisited" what if we moved finalized nodes and their distances into a separate dictionary?

Rewrite the `cost_from_node` function in `graph.py`:

```
            # Visited nodes are already minimized, skip them
              if v in finalized_dist:
                  continue

              # What is the cost to this neighbor?
              alt = current_node_cost + edge.cost

              # Is this the first time I am seeing the node?
              if v not in seen_dist:

                  # Insert into the seen_dict, prev, and priority queue
                  seen_dist[v] = alt
                  prev[v] = current_node
                  pqueue.add(v, alt)

              else: # v has been seen. Is this a cheaper route?
                  old_dist = seen_dist[v]
                  if alt < old_dist:
                      # Update the seen_dict, prev, and priority queue
                      seen_dist[v] = alt
                      prev[v] = current_node
                      pqueue.update(v, old_dist, alt)

        return (finalized_dist, prev)
```

This should be have exactly the same except for the unreachable nodes. If you have a graph that is not connected, there will be nodes that can't be reached from the origin. In the old version, these had a cost of `math.inf`. Now they just won't be in the dictionary at all. So, change `cities.py` to deal with this:

```
if nyc in cost_from_long_beach:
    nyc_cost = cost_from_long_beach[nyc]
    print(f"\n*** Total cost from Long Beach to NYC: ${nyc_cost:.2f} ***")

    path_to_nyc = graph.shortest_path(prev, nyc)
    print(f"\n*** Cheapest path from Long Beach to NYC: {path_to_nyc} ***")
else:
    print("You can't get to NYC from Long Beach")
```

If you run `cities.py` now, it should behave exactly like the old version.

But there is a bug. It will rear its head if two cities with the same cost are in the priority queue together. Change `cities.py` so that Denver and Pheonix have the same cost:

```
graph.WeightedEdge(12, long_beach, los_angeles)
graph.WeightedEdge(19, los_angeles, denver)
graph.WeightedEdge(19, los_angeles, pheonix)
```

Now try running it. You should get an error:

```
TypeError: '<' not supported between instances of 'Node' and 'Node'
```

What happened? The `loc_for_pair` method is comparing tuples made up of a `float` and a `Node`. The `float` comes first in the tuple, so that is compared first. However, if the two tuples have the same priority, it then compares nodes.

The error statement say "Nodes don't have a less-than method; I don't know how to compare them."

Each `Node` lives at an address in memory. You can get that address as a number using the `id` function. The ID is unique and constant over the life of the object. It is a rather arbitary ordering, but it will work for this problem. Add a method to your `Node` class:

```
    # Nodes will be ordered by their location in memory
    def __lt__(self, other):
        return id(self) < id(other)
```

Fixed.

Now let's make the priority queue more efficient.

1.3 Binary Search

The phone company in every town used to print a thing called a phone book. The names and phone numbers were arranged alphabetically. As you might imagine, these books often had more than a thousand pages.

If you were looking for “John Jeffers”, you wouldn’t start at the first page and read sequentially until you reached his name. You would open the book in the middle, and see a name like “Mac Miller”, and then think “Jeffers comes before Miller”. Then you would split the pages in your left hand in half and see a name like “Hester Hamburg” and think “Jeffers comes after Hamburg”. Then you would split the pages in your right hand, and so on until you found the page with “John Jeffers” on it.

That is binary search.

Binary Search is a search algorithm that finds the position of a target value within a sorted array. The binary search algorithm works by repeatedly dividing the search interval in half. If the target value is equal to the middle element of the array, the position is returned. If the target value is less or greater than the middle element, the search continues in the lower or upper half of the array respectively.

1.4 Algorithm

The binary search algorithm can be described as follows:

1. If the array is empty, the search is unsuccessful, so return ”Not Found”.
2. Otherwise, compare the target value to the middle element of the array.
3. If the target value matches the middle element, return the middle index.
4. If the target value is less than the middle element, repeat the search with the lower half of the array.
5. If the target value is greater than the middle element, repeat the search with the upper half of the array.
6. Repeat steps 2-5 until the target value is found or the array is exhausted.

```
class PriorityQueue:
    def __init__(self):
        self.list = []

    # Return and remove the first item
    def pop(self):
        if len(self.list) > 0:
            return self.list.pop(0)
        else:
            return None

    def __len__(self):
        return len(self.list)

    def add(self, value, priority):
        pair = (priority, value)
        i = self.loc_for_pair(pair)
        self.list.insert(i, pair)

    def update(self, value, old_priority, new_priority):
        old_pair = (old_priority, value)
        i = self.loc_for_pair(old_pair)
        del self.list[i]
        self.add(value, new_priority)

    def loc_for_pair(self, pair):
        # The range where it could be is [lower, upper)
        # Start with the whole list
        lower = 0
        upper = len(self.list)

        while upper > lower:
            next_split = (upper + lower) // 2
            v = self.list[next_split]
            if pair < v:  # pair is to the left
                upper = next_split
            elif pair > v:  # pair is to the right
                lower = next_split + 1
            else: # Found pair!
                return next_split
        return lower
```

If you try running it now, it should work perfectly.

Now you have a graph class that would find the cheapest path quickly even if it had thousands of nodes with thousands of edges.

Appendix A

Answers to Exercises

Index

Binary Search, 5

Priority Queue, 1