Algorithms and Complexity

Unit 2

David Emanuel Craciunescu

Exercise 1

For an even-length non-empty array of non-repeating pseudo-randomly-generated positive integer values, form unique pairs between such values and return the pair that yields the highest result when both its numbers are added together.

Design a greedy algorithm that creates pairs in such a way that the maximum value of the sums is the smallest *biggest* sum possible, showing that the candidate selection function used provides an optimal solution.

Solution

Presorting the array and then looking for elements starting at opposite ends seemed to be like a great solution. We can also only iterate half of the array since once we reach the midpoint all pairs have already been explored and found. With this we calculate the complexity the following way:

- Sort the array O(n log(n)) as per TimSort's Python standard worst case complexity.
- Iterate over half of it and make pairs from start and end => O(n/2) => O(n)
- Store biggest value found each iteration O(1)
- Return stored value O(1)

We would obtain a resulting **time complexity of O(nlog(n))** and a **space complexity of O(n)**.

Implementation

```
def get_min_pair_sum(values):
    # Initialization and preprocessing
    current_max = []
    values.sort()

    size = len(values)
    midpoint = size // 2
    iterable_values = enumerate(values[:midpoint])

    # Iterate over array.
    for idx, val in iterable_values:

        start = val
        end = values[size-idx-1]
        new_pair = [start, end]

        # Check for max pair.
        new_max_found = sum(new_pair) > sum(current_max)
        if new_max_found:
            current_max = new_pair

    return sum(current_max)
```

Tests

The tests are as simple as they can get. Check that the result is the smallest possible big sum.

```python
def test_get_min_pair_sum01(self):
    self.assertEqual(get_min_pair_sum([5, 8, 4, 1, 7, 9]), 12)

def test_get_min_pair_sum02(self):
    self.assertEqual(get_min_pair_sum([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]), 11)

def test_get_min_pair_sum03(self):
    self.assertEqual(get_min_pair_sum([7, 7, 7, 7, 7, 7]), 14)

def test_get_min_pair_sum04(self):
    self.assertEqual(get_min_pair_sum([1000, 1, 1234, 3, 23455, 4, 6666, 9]), 23456)
```

```
┌dave💻ec2 📁 algo 🌿master ❯ python3 -m unittest U2/test/test_e1.py
----------------------------------------------------------------------
Ran 4 tests in 0.000s | OK
```

Exercise 2

We are tasked with storing n files on a magnetic tape (sequential path storage medium) in such a way that the average loading time for each of the files is minimized. In the magnetic tape, each time data is requested, the system will search the tape sequentially and then rewind back to the beginning after finishing the operation.

Assume that along with the files, we are provided with their lengths and the amount of times the file has been searched for in the past. Assume the reading speed and the information density within the tape are constant.

Design a greedy algorithm that minimizes the average loading time when searching for data in the magnetic tape. Analyze the efficiency and complexity for the provided solution.

Solution

Given that we want to optimize for average loading time, we'll have to sort files by a function of their length and their use. We've decided to simply divide lenght and the amount of times they're requested. With this we sort the files according to that criteria. A simple python lamba function can solve the problem for us.

Implementation

```python
def store_files(files):
    return sorted(files, key=lambda elem: elem[0] / elem[1])
```

Tests

The tests here were quite fun. The exercise itself seemed too simple so we decided to monkey test it. The tests work thanks to:

- A "calculate_cost" method that manually calculates the cost associated with each possible test solution. Its code is extremely straightforward.
- A "generate_vector" method that takes a seed and creates a random dataset for us that emulates the datasets we could find.
- A "test_general" method that generates all the permutations of the dataset we generate and returns the one whose cost is the smallest.

```python
def calculate_cost(files):
    """ Calculate the cost associated with a possible solution """
    offset = cost = 0

    for file in files:
        cost += (offset + file[0]) * file[1]
```

```
        offset += file[0]

    return cost

def generate_vector(seed):
    """ Generate a test vector """
    r.seed(a=seed)
    max_len = r.randint(1, 10)
    return [[r.randint(1, 100), r.randint(1, 100)] for _ in range(1, max_len)]

def test_general(vector):
    """ Generate vector permutations and return minimum cost """
    perms = list(permutations(vector))
    return min([calculate_cost(perm) for perm in perms])
```

The actual tests themselves are very straightforward. Check our provided result against what we expect to receive.

```
def test_store_files01(self):
    test_vector = generate_vector("test_store_files01")
    provided = calculate_cost(store_files(test_vector))
    expected = test_general(test_vector)

    self.assertEqual(provided, expected)
```

Here we can make sure it all checks out.

```
┌dave💻ec2 📁 algo 🌿master ❯ python3 -m unittest U2/test/test_e2.py
----------------------------------------------------------------------
Ran 4 tests in 1.740s | OK
```

Exercise 3

For a non-emtpy vector of length n, find the minimum and maximum element of the vector. The type of data the vector contains is not relevant to the problem, but performing comparisons between elements in the vector is an extremely expensive operation, so you should minimize the amount of comparisons.

Implement a greedy method that makes a maximum of (3/2)n comparisons. Analyze the efficiency and complexity for the provided solution.

Solution

By using the Tournament Method, we compare elements in pairs through to the next stage, this can only leave one element at the end. With this method we obtain a max of comparisons that follow the formula:

- if n is odd => 3(n-1)/2
- if n is even => 3(n/2)-2

In either case both are upper bounded by (3/2)n comparisons.

The time complexity of the solution is clearly O(n), since we iterate once over the dataset.

Implementation

```
def min_and_max(vector):
```

```
    if not vector:
        return []

    length = len(vector)
    pair = [vector[0], vector[1]]
    even_len = length % 2 == 0

    # Make sure we start in the right place
    if even_len:
        min_current, max_current = min(pair), max(pair)
        idx = 2
    else:
        min_current = max_current = vector[0]
        idx = 1

    # Iterate while looking for candidates
    while idx < length - 1:
        current_val, next_val = vector[idx], vector[idx+1]
        found_new_minimum = current_val < next_val

        if found_new_minimum:
            max_candidate, min_candidate = next_val, current_val
        else:
            max_candidate, min_candidate = current_val, next_val

        max_current = max(max_current, max_candidate)
        min_current = min(min_current, min_candidate)

        idx += 2

    return [max_current, min_current]
```

Tests

The tests are pretty staightforward here as well. We offer datasets that match with the description of the exercise and check we obtain the expected value.

```
    def test_min_and_max01(self):
        provided = min_and_max([1, 2, 3, 4, 5])
        expected = [5, 1]
        self.assertEqual(provided, expected)

    def test_min_and_max02(self):
        provided = min_and_max([5, 2, 6, 1, 2, 4, 6])
        expected = [6, 1]
        self.assertEqual(provided, expected)

    def test_min_and_max03(self):
        provided = min_and_max([-1, -5, 2, 2, 4, 5])
        expected = [5, -5]
        self.assertEqual(provided, expected)

    def test_min_and_max04(self):
```

```
        provided = min_and_max([])
        expected = []
        self.assertEqual(provided, expected)
```

```
┌dave💻ec2 📁 algo 🌿master ❯ python3 -m unittest U2/test/test_e3.py
----------------------------------------------------------------------
Ran 4 tests in 0.000s | OK
```

Exercise 4

You are given a non-directed graph G = <N, A>, where N = {1, ..., n} is the set of nodes and A ⊆ NxN is the set of edges. Each edge (i,j) ∈ A has an associated cost Cij (Cij > ∀i, j ∈ N; if (i,j) ∉ A can be considered Cij =+ ∞). Let M be the cost matrix of the graph G, that is, M[i,j] = Cij (since the non-directed graph is that (i,j) = (j,i) so the matrix M is symmetric).

Having as data (1) the number of nodes n, and (2) the cost matrix M, find the minimum support tree of a graph G using Prim's algorithm. For your implementation, consider the following ideas:
- Unlike Kruskal's algorithm (which creates the tree using independent connected components that are joined together, Prim's algorithm is based on the idea of building an increasingly large, starting with a single node and ending by coating the entire graph.
- The algorithm begins with a tree of a node, to which a second node is added, then a third node, etc., until the n nodes are joined together. The way to choose a node is looking for the nearest node to the whole tree, without creating cycles.
- As the size of the tree grows, the search for the nearest node becomes complicated. For the algorithm to be efficient, take into consideration the following guidelines:

- **Create a data structure to store the best distance from each node to the set of nodes of the tree.**
- **Do not exceed a time complexity of O(n²).**
- **Storing how the tree was created is necessary, for example, by indicating to which node of the tree the new candidate is being joined.**

Analyze the efficiency and complexity of the provided solution.

Solution

The data structure that was used for this implementation was a priority queue through the use of a minheap thanks to the standard python implementation.

With this, we can standarize the format of an edge (weight, start, end) in a tuple, and the heap itself will take care to always give us back the edge that is closest in costant time.

The time complexity of the provided solution is, as per the theoretical definition of Prim's Algorithm, O((V+E) log V), where V is the number of vertices and E is the number of edges. This happens because each vertex is inserted in the priority queue only once, and the insertion in a prio queue takes logarithmic time.

I've taken extra care to make the code **as clean and as straightforward as possible** knowing that explaining it on paper would be practically impossible. If by any chance it is still not clear enough, contact me and I'll gladly modify this section and attach an explanation.

Implementation

```
def prim(graph, start):
    # Check empty input.
    if not graph:
        return {}

    # Initialize path in tree and visited nodes.
```

```
    min_span_tree = defaultdict(set)
    visited = set([start])

    # Get all edges from start point.
    # << Python implement minheap by default >>
    neighbors = graph[start].items()
    edges = [(weight, start, end) for end, weight in neighbors]
    heap.heapify(edges)

    while edges:
        weight, start, end = heap.heappop(edges)

        # Check for cycles.
        if end not in visited:
            visited.add(end)
            min_span_tree[start].add(end)

            # Add edges of the newly visited node.
            neighbors = graph[end].items()
            for next_node, weight in neighbors:
                # Check for cycles.
                if next_node not in visited:
                    heap.heappush(edges, (weight, end, next_node))

    return min_span_tree
```

Tests

Here's one of the tests to make sure it works.

```
def test_prim01(self):
        graph = {
            'A': {'B': 0.5, 'C': 3, 'E': 0},
            'B': {'A': 0.5, 'E': 0.7},
            'C': {'A': 3, 'D': -4.1, 'E': 2},
            'D': {'C': -4.1, 'F': 11},
            'E': {'A': 0, 'B': 0.7, 'C': 2, 'D': 1, 'F': 7, 'G': -1},
            'F': {'D': 11, 'E': 7, 'J': 3},
            'G': {'E': -1, 'H': 4, 'I': -1, 'J': 0.2},
            'H': {'H': 4},
            'I': {'G': -1, 'J': -2},
            'J': {'F': 3, 'G': 0.2, 'I': -2}
        }

        provided = dict(prim(graph, 'A'))

        expected = {
            'A': {'B', 'E'},
            'E': {'D', 'G'},
            'G': {'H', 'I'},
            'I': {'J'},
            'D': {'C'},
            'J': {'F'}
```

```
        }

        self.assertEqual(provided, expected)
```

And here's evidence that the tests actually check out.

```
┌dave💻ec2 📁 algo 🌿master ❯ python3 -m unittest U2/test/test_e4.py
----------------------------------------------------------------------
Ran 3 tests in 0.000s | OK
```

Exercise 5

You are given a non-directed graph G = <N, A>, where N = {1, ..., n} is the set of nodes and A ⊆ NxN is the set of edges. Each edge (i,j) ∈ A has an associated cost Cij (Cij > ∀i, j ∈ N; if (i,j) ∉ A can be considered Cij =+ ∞). Let M be the cost matrix of the graph G, that is, M[i,j] = Cij (since the non-directed graph is that (i,j) = (j,i) so the matrix M is symmetric).

Having as data (1) the number of nodes n, and (2) the cost matrix M, find the minimum path between nodes 1 and n, and the length of said path using the Dijkstra algorithm. When implementing the algorithm, take into consideration the following ideas:
- Create or make use of a data structure that stores the known temporal distances for the vertices the algorithm hasn't traveled to yet.
- Select as candidate the one with the least known temporal distance, eliminate it from the set of vertices not traveled to before, and update the rest of the temporal distances if they can be improved using the current vertex.
- Store the way in which the graph was traversed from vertex 1 to vertex n (not necessarily equal to the set of decisions taken).

Analyze the efficiency and complexity of the provided solution.

Solution

Pretty much the same schema used for the previous exercise was used for this one, only here we're taking into consideration distances instead of looking for a spanning tree.

The explanation is exactly the same one, and with the same data structure, we obtain a very similar complexity. In this case it is O(V + ElogV, as per the theoretical specification, where V is the amount of vertices and E is the amout of edges.

For some reason I haven't been able to implement Dijkstra as the own exercise defines. Here's a generic Dijkstra implementation, even though it's not exactly what the problem asks for.

Implementation

```
def path_shortest(graph, start):
    # Initialize all distances to infinity but the start one.
    distances = {node: float('infinity') for node in graph}
    distances[start] = 0
    paths = [(0, start)]

    while paths:
        current_distance, current_node = heap.heappop(paths)
        neighbors = graph[current_node].items()

        for neighbor, weight in neighbors:
            distance = current_distance + weight
```

```
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heap.heappush(paths, (distance, neighbor))

    return distances
```

Exercise 6

Shrek, Donkey and Dragona have just arrived to the base of Lord Farquaad's towering castle determined to free Fiona from her confinement. As they suspected, the drawbridge is guarded by multiple soldiers. Worry not, for they suspected such a thing could happen and carried with them a huge amount of ladders of all kinds and lengths so that they could get over the wall easily.

Unfortunately, no ladder would be enough for them to get over the wall. Dragona is also unable to fly that high, so they seemed to be stuck at first. But just when they were about to give up, Shrek realized that the ladders they brought with them were all made of iron, which could be easily welded by Dragona's fire.

She can weld any two ladders with her fire breath, but the time it takes for them to be completely welded is equal to the sum of their length. For example, welding a 6-meter ladder and an 8-meter one would take 6+8 = 14 units of time. If this ladder were then welded to a 7-meter ladder, the time it would take to weld them would be 14+7 = 21 units of time. With this approach, it would have taken 14+21 = 35 units of time to complete the ladder.

Design an efficient algorithm that finds the best cost and way to weld the stairs so that Shrek takes as little time as possible to get past the wall. Indicate the data structures that were used to solve this exercise and justify their use. Assume the set of ladders they have always contains only the exact amount of ladders needed to get past the wall.

Analyze the efficiency and complexity of the provided solution.

Solution

Just like the previous two exercises. Once you discover what a priority queue can do, there are few things that cannot benefit from it. In this case the provided vector is heapified in order to sort it and then the smallest two values are always added together (keeping count of the time elapsed while welding) and then added back to the heap.

We obtain a very elegant complexity of O(nlogn), given that we iterate over all the elements once (O(n))

Implementation

```
def minimum_sum(vector):
    time = 0

    # Check for input.
    if vector:
        heap.heapify(vector)

        # Add two smallest elements repeatedly.
        while vector:
            weld = heap.heappop(vector) + heap.heappop(vector)
            if vector:
                heap.heappush(vector, weld)

            time += weld

    return time
```

Tests

```
def test_minimum_sum01(self):
    provided = minimum_sum([1, 2, 3, 4, 5])
    expected = 33
    self.assertEqual(provided, expected)

def test_minimum_sum02(self):
    provided = minimum_sum([])
    expected = 0
    self.assertEqual(provided, expected)

def test_minimum_sum03(self):
    provided = minimum_sum([-1, -1, -1, -1, -1])
    expected = -14
    self.assertEqual(provided, expected)

def test_minimum_sum04(self):
    provided = minimum_sum([0, 0, 0, 0, 13])
    expected = 13
    self.assertEqual(provided, expected)
```

```
┌dave💻ec2 📂 algo 🌿master ❯ python3 -m unittest U2/test/test_e6.py
----------------------------------------------------------------------
Ran 4 tests in 0.000s | OK
```
