```python
# coding=utf-8

"""
Unit tests for key handling
"""

import os
from unittest import TestCase

from src.crypto import keys


class TestRSAKeyLoading(TestCase):
    """Tests for RSA Keys"""

    def setUp(self) -> None:
        """Initialise loaders fresh between tests"""
        self.loader = keys.RSAKeyLoader()
        os.chdir("/home/tcassar/projects/settle/src")
        print(os.system("pwd"))
        self.key_path = "./crypto/sample_keys/d_private-key.pem"
        self.pub_key_path = "./crypto/sample_keys/d_public-key.pe"

    def test_file_loading(self):
        """Tests that file is being loaded correctly assuming correct file"""

        expected_start = "modulus"
        # test assumes if it starts off fine it will continue being fine
        received_start = self.loader.load(self.key_path)
        self.assertEqual(expected_start, received_start[:7])

    def test_parsing(self):
        """Checks that keys are parsed correctly and that errors are raised when inputs are incorrect"""
```

```
34          loader = self.loader
35
36          loader.load(self.key_path)
37          loader.parse()
38
39          # compare parsed file to known values from
    test keys.
40          # lay out known values
41          k_e = 65537
42
43          k_d =
    42318605026103475553526985044230475639442568138229963925734038989576569874864579669508371562955146126547877955291914430323301128393346172685767442683818738967216638476720764074944647817540070989695696588210951082558372250397158894945714319204774976709066583095965638944919595239862657801000401988151966317962866310493446444221800190039054548479140813630569444720847689992964879750244948034517871603890465032993904165597478235001604038074703682680413388703757501362444680122222701317562224831854480644662034354542275281227314387046825525570376824278684460211954070416261823570911887469407283866423584255451231566190543
44          k_n =
    21616792031143752746309415579452320202510893126073087652383723408105063600070147761500936454570470862786970206983368636166003008975173251124763374054321346030354457021425165356037781636930036106404418295413726431002556836900067049867944499043128421557451025437279819137427553642625019821057158620227234339857381390676943784764665144556770107968670906167894855774586373708692617743377046549318417755362244202417503903181827421441408785602795322818434551136750209831842525036691384924518828835962027759009110005026961294337702970618450353224231935687225848095890069620605874219546032017844978461582635821441774306426043
45
46          # build lists for testing
47          cases = [
48              "pub_exp",
49              "priv_exp",
```

```
50              "mod",
51              "prime1",
52              "prime2",
53              "exponent1",
54              "exponent2",
55              "coefficient",
56          ]
57
58          known = [k_e, k_d, k_n]
59
60          key = keys.RSAPrivateKey(loader)
61
62          received = [
63              key.e,
64              key.d,
65              key.n,
66              key.p,
67              key.q,
68              key.exp1,
69              key.exp2,
70              key.crt_coef,
71          ]
72
73          for test, known_val, rec_val in zip(cases,
    known, received):
74              with self.subTest(test):
75                  self.assertEqual(rec_val, known_val)
76                  self.assertEqual(type(rec_val), type
    (known_val))
77
78      def test_file_not_found(self):
79          path = "./adsads"
80
81          with self.assertRaises(keys.RSAParserError):
82              loader = self.loader
83              loader.load(path)
84
85      def test_wrong_format(self):
86          """Checks that we can deal with files being
    the wrong format"""
87          with self.assertRaises(keys.RSAParserError):
```

```
 88             self.loader.load("./crypto/sample_keys/
    d_public-key.pe")
 89
 90     def test_unparsed_key(self):
 91         """Check accessing attributes"""
 92         loader = self.loader
 93         loader.load(self.key_path)
 94         key = keys.RSAPrivateKey(loader)
 95
 96         with self.assertRaises(keys.RSAKeyError):
 97             _ = key.asdf
 98
 99     def test_unloaded_key(self):
100         """Checks that error is raised if parse()
    is called on a loader object which has not yet
    loaded a key"""
101         ldr = self.loader
102         with self.assertRaises(keys.RSAParserError
    ):
103             ldr.parse()
104
105     def test_public_key(self):
106         """Checks that public keys will raise an
    exception if they are used for signing"""
107         ldr = self.loader
108         ldr.load(self.key_path)
109         ldr.parse()
110
111         pub_key = keys.RSAPublicKey(ldr)
112
113         with self.subTest("allowed access"):
114             self.assertEqual(pub_key.e, 65537)
115
116         with (self.subTest("deny access"), self.
    assertRaises(keys.RSAPublicKeyError)):
117             _ = pub_key.p
118
```

```python
# coding=utf-8
import hashlib
import sys
from unittest import TestCase

from src.crypto import hashes


class TestHash(TestCase):
    """
    Test hash interfaces (functionality not aim of testing as functionality comes from hashlib
    1) Check initialisation as expected
    2) Check update
    3) Check digest
    4) Check hexdigest == intdigest
    """

    def test_init(self) -> None:
        """Checks that _hasher is initialised correctly i.e. no strange start values"""
        h = hashes.Hasher()
        self.assertEqual(
            h.digest().int_digest(),
            int.from_bytes(hashlib.sha3_256(b"").digest(), byteorder=sys.byteorder),
        )  # should be initialised empty

    def test_hash(self) -> None:
        """tests that hash object can validate hash looking numbers"""
        with self.subTest("valid"):
            h_val = hashlib.sha3_256(b"1234").digest()
            h = hashes.Hash(h_val)  # create a definitely valid hash

        with self.subTest("too short"):
            with self.assertRaises(hashes.HashError):
                hashes.Hash(b"12")
```

```
35
36          with self.subTest("too long"):
37              with self.assertRaises(hashes.HashError
    ):
38                  hashes.Hash(
39                      b"
    1157920892373161954235709850086879078532699846656405
    6403945758400791312963993 60"
40                  )
41
42          with self.subTest("wrong type"):
43              with self.assertRaises(hashes.HashError
    ):
44                  # noinspection PyTypeChecker
45                  hashes.Hash("dave")
46
47      def test_update_digest(self) -> None:
48          """Ensures that a hash with a given value
    will digest the correct thing"""
49          h = hashes.Hasher(b"1234")
50          h_ = hashlib.sha3_256(b"1234")
51
52          with self.subTest("before update"):
53              self.assertEqual(h.digest().h, h_.digest
    ())
54
55          update_msg = b"test hash update"
56          h.update(update_msg)
57          h_.update(update_msg)
58
59          with self.subTest("after update"):
60              self.assertEqual(h.digest().h, h_.digest
    ())
61
62      def test_hasher_fails(self):
63          """Checks that hasher objects when not bytes
     are passed into it"""
64          with self.assertRaises(hashes.HasherError):
65              # noinspection PyTypeChecker
66              hasher = hashes.Hasher("abd")
67
```

```python
# coding=utf-8

"""
Testing sign / verify through RSA working as expected
"""
import os
from unittest import TestCase

from src.crypto import keys
from src.crypto import rsa
from src.transactions.transaction import Signable


def setUpModule():
    os.chdir("/home/tcassar/projects/settle/src")


class TestRSA(TestCase):
    """Just tests RSA parts"""

    def setUp(self) -> None:
        # load keys
        ldr = keys.RSAKeyLoader()
        ldr.load("./crypto/sample_keys/d_private-key.pem")
        ldr.parse()

        # build public and private
        self.private = keys.RSAPrivateKey(ldr)
        self.public = keys.RSAPublicKey(ldr)

    def test_encryption(self):
        """Checks to see if process is reversible, and encrypted is different to how it started"""
        message = " | maia"
        m_bytes = bytes(message, encoding="utf8")
        encrypted = rsa.RSA.encrypt(m_bytes, self.public)
        # TODO: actually fix byte overflow affair
        decrypted = rsa.RSA.bytes_to_str(rsa.RSA.
```

```
37 naive_decrypt(encrypted, self.private))
38
39         with self.subTest("Catch Public Key"):
40             with self.assertRaises(rsa.
   DecryptionError):
41                 _ = rsa.RSA.naive_decrypt(encrypted
   , self.public)  # type: ignore
42
43         with self.subTest("encrypted"):
44             self.assertNotEqual(m_bytes, encrypted)
45
46         with self.subTest("Successful decryption"):
47             self.assertEqual(message, decrypted)
48
49     def test_RSA_sign(self):
50         """Checks for consistent creating /
   verifying of a 'signature'"""
51
52         message = " | maia"
53         m_bytes = message.encode("utf8")
54         sig: bytes = rsa.RSA.sign(m_bytes, self.
   private)
55         de_sign: bytes = rsa.RSA.inv_sig(sig, self.
   public)
56
57         self.assertEqual(m_bytes, de_sign)
58
```

```python
# coding=utf-8
from unittest import TestCase

from src.simplify.base_graph import Digraph
from src.simplify.flow_graph import FlowGraph
from src.simplify.graph_objects import Vertex
from src.simplify.path import Path, prev_map, disc_map, BFSQueue
from src.simplify.weighted_digraph import WeightedDigraph


class TestPath(TestCase):
    """Tests searching for BFS"""

    def setUp(self):
        # make a graph with 5 nodes

        labels = ["a", "b", "c", "d", "e", "f"]
        self.vertices = [Vertex(ID, label=label) for ID, label in enumerate(labels)]

        self.g = Digraph(self.vertices)
        a, b, c, d, e, f = self.vertices
        self.g.add_edge(a, b, c)
        self.g.add_edge(b, d)
        self.g.add_edge(d, f)
        self.g.add_edge(c, e)
        self.g.add_edge(d, f)
        self.g.add_edge(e, f, b)

        # set up weighted_graph and flow graphs
        self.weighted_graph = WeightedDigraph(self.vertices)
        self.flow_graph = FlowGraph(self.vertices)

        for g in [self.weighted_graph, self.flow_graph]:
            g.add_edge(a, (b, 10), (c, 10))
            g.add_edge(b, (d, 25))
            g.add_edge(c, (e, 25))
```

```
37              g.add_edge(d, (f, 10))
38              g.add_edge(e, (f, 10), (b, 6))
39
40      def test_shortest_path(self):
41          """
42          Checks that we can in fact find shortest
    path along
43          """
44
45          a, b, c, d, e, f = self.vertices
46
47          cases = ["digraph", "weighted_graph", "flow"
    ]
48
49          # check precalulated shortest path
50          graph_cases = [self.g, self.weighted_graph,
    self.flow_graph]
51
52          for case, g in zip(cases, graph_cases):
53              with self.subTest(case):
54                  calc_shorted: list[Vertex] = Path.
    shortest_path(
55                      self.g, a, f, self.g.neighbours
56                  )
57                  expected: list[Vertex] = [a, c, e, f
    ]
58                  self.assertEqual(expected,
    calc_shorted)
59
60      def test_build_path(self):
61          """Given a prev_map, check we build the
    right path"""
62
63          # generate vertices, unpack into vars
64          vertices = []
65          for ID, name in enumerate(["A", "B", "C", "D
    ", "E", "F"]):
66              vertices.append(Vertex(ID, name))
67
68          a, c, b, d, e, f = vertices
69
```

```
 70         # build a prev map. for context, path is A
    -> B -> D -> F
 71         prev: prev_map = {a: None, b: a, c: a, d: b
    , e: c, f: d}
 72         expected = [a, b, d, f]
 73
 74         self.assertEqual(expected, Path._build_path
    (prev, f))
 75
 76     def test_recursive_bfs(self):
 77         """Check that BFS is finding paths along
    graph correctly"""
 78         graph = self.flow_graph
 79         a, b, c, d, e, f = graph.nodes()
 80
 81         expected: prev_map = {a: None, b: a, c: a,
    d: b, e: c, f: e}
 82
 83         # set up structures for BFS
 84         # create queue, discovered list, previous
    list
 85         queue = BFSQueue(next(iter(graph.graph)))
 86
 87         discovered: disc_map = {node: False for
    node in graph.nodes()}
 88         prev: prev_map = {node: None for node in
    graph.nodes()}
 89
 90         self.assertEqual(
 91             expected,
 92             Path.BFS(
 93                 graph=graph,
 94                 queue=queue,
 95                 discovered=discovered,
 96                 target=None,
 97                 previous=prev,
 98                 neighbours=graph.neighbours,
 99             ),
100         )
101
102     def test_find_target(self):
```

```
103         """Checks that BFS can stop when its given
    target is found"""
104         graph = self.flow_graph
105         a, b, c, *_ = graph.nodes()
106 
107         expected = {node: None for node in graph.
    nodes()}
108         expected[b] = a
109         expected[c] = a
110 
111         queue, discovered, previous = Path.
    build_bfs_structs(graph, a)
112 
113         calculated = Path.BFS(
114             graph=graph,
115             queue=queue,
116             discovered=discovered,
117             previous=previous,
118             target=b,
119             neighbours=graph.neighbours,
120         )
121         self.assertEqual(expected, calculated)
122 
123     def test_build_bfs_struct(self):
124         """Checks that structures that are built
    for BFS are built correctly"""
125         with self.subTest("with initial value"):
126             queue, disc, prev = Path.
    build_bfs_structs(
127                 self.flow_graph, self.vertices[0]
128             )
129             self.assertEqual(queue, BFSQueue(self.
    vertices[0]))
130 
131         with self.subTest("with initial value"):
132             queue, disc, prev = Path.
    build_bfs_structs(self.flow_graph)
133             self.assertEqual(queue, BFSQueue())
134 
```

File - /home/tcassar/projects/settle/tests/test_settling/test_flow.py

```
 1 # coding=utf-8
 2 from unittest import TestCase
 3
 4 from src.simplify.flow_algorithms import
   NoOptimisations, MaxFlow, Simplify
 5 from src.simplify.flow_graph import *
 6 from src.simplify.graph_objects import Vertex
 7
 8
 9 class TestFlowEdge(TestCase):
10     """Tests for FlowEdge"""
11
12     def setUp(self) -> None:
13         self.edges = [FlowEdge(Vertex(0), Vertex(1
   ), 5 * n) for n in range(2)]
14         self.edges[0].flow = -3  # => unused
   capacity should be three
15
16     def test_unused_capacity(self):
17         """builds two edges, residual and non-
   residual
18         non-residual has capacity 5"""
19
20         for edge, cap in zip(self.edges, [3, 5]):
21             with self.subTest(edge):
22                 self.assertEqual(edge.
   unused_capacity(), cap)
23
24     def test_push_flow(self):
25         """Checks that we update flow by required
   amount"""
26         # push three units of capacity through each
   edge
27         for edge, exp in zip(self.edges, [0, 2]):
28             with self.subTest(edge):
29                 edge.push_flow(3)
30                 self.assertEqual(exp, edge.
   unused_capacity())
31
32         # try to push excess amount of flow down an
   edge
```

```
33          with self.subTest("exceed capacity"), self.
   assertRaises(FlowEdgeError):
34              self.edges[1].push_flow(3)
35
36      def test_adjust_edge(self):
37          """Tests that the adjust_edges function
   works as expected
38          i.e when adjust is called, edges in the
   graph morph to edges with a flow equal to their
   unused capacity
39          and all edges of weight 0 are deleted"""
40          (
41              res,
42              fwd,
43          ) = self.edges
44          fwd.push_flow(3)
45          res.push_flow(-3)
46
47          new_fwd = FlowEdge(Vertex(0), Vertex(1), 2)
48          new_res = FlowEdge(Vertex(0), Vertex(1), 0)
49
50          fwd.adjust_edge()
51          res.adjust_edge()
52
53          self.assertEqual(new_fwd, fwd)
54          self.assertEqual(new_res, res)
55
56          # catch EdgeCapacityZero
57          with self.assertRaises(EdgeCapacityZero):
58              fwd.push_flow(2)
59              fwd.adjust_edge()
60
61
62  class TestFlowGraph(TestCase):
63      def setUp(self) -> None:
64          # make some nodes for a graph
65          self.nodes = [Vertex(n, label=chr(n + 97))
   for n in range(5)]
66          self.graph = FlowGraph(self.nodes)
67
68      def test_is_edge(self):
```

```
 69          a, b, c, d, e = self.nodes
 70
 71          with self.subTest("no edge"):
 72              self.assertFalse(self.graph.is_edge(a,
    b))
 73
 74          with self.subTest("edge"):
 75              self.graph.add_edge(a, (b, 4))
 76              self.assertTrue(self.graph.is_edge(a, b
    ))
 77
 78      def test_get_edge(self):
 79          a, b, c, d, e = self.graph.nodes()
 80          self.graph.add_edge(a, (b, 5))
 81          self.assertEqual(FlowEdge(a, b, 5), self.
    graph.get_edge(a, b))
 82
 83      def test_add_edge(self):
 84          a, b, c, d, e = self.nodes
 85          # check there aren't edges
 86          with self.subTest("negative test"):
 87              self.assertFalse(self.graph.is_edge(a,
    b))
 88              self.assertFalse(self.graph.is_edge(b,
    a, residual=True))
 89          self.graph.to_dot()
 90          self.graph.add_edge(a, (b, 5))
 91          self.graph.to_dot()
 92          with self.subTest("added"):
 93              self.assertTrue(self.graph.is_edge(a, b
    ))
 94              self.assertTrue(self.graph.is_edge(b, a
    , residual=True))
 95              self.assertFalse(self.graph.is_edge(b,
    a))
 96
 97          with self.subTest("Net debt (adding)"):
 98              self.assertEqual(
 99                  {
100                      Vertex(ID=0, label="a"): 5,
101                      Vertex(ID=1, label="b"): -5,
```

```
102                         Vertex(ID=2, label="c"): 0,
103                         Vertex(ID=3, label="d"): 0,
104                         Vertex(ID=4, label="e"): 0,
105                     },
106                     self.graph.net_debt,
107                 )
108 
109             with self.subTest("Net debt (removing)"):
110                 self.graph.pop_edge(a, b, update_debt=
    True)
111                 self.assertEqual(
112                     {
113                         Vertex(ID=0, label="a"): 0,
114                         Vertex(ID=1, label="b"): 0,
115                         Vertex(ID=2, label="c"): 0,
116                         Vertex(ID=3, label="d"): 0,
117                         Vertex(ID=4, label="e"): 0,
118                     },
119                     self.graph.net_debt,
120                 )
121 
122         def test_pop_edge(self):
123             a, b, c, d, e = self.nodes
124 
125             self.graph.add_edge(a, (b, 5))
126 
127             with self.subTest("negative"):
128                 self.assertTrue(self.graph.is_edge(a, b
    ))
129                 self.assertTrue(self.graph.is_edge(b, a
    , residual=True))
130 
131             self.graph.pop_edge(a, b)
132 
133             with self.subTest("removed edge"):
134                 self.assertFalse(self.graph.is_edge(a,
    b))
135             with self.subTest("removed residual"):
136                 self.assertFalse(self.graph.is_edge(b,
    a, residual=True))
137 
```

```
138     def test_flow_neighbours(self):
139         """Checks we get edges that have unused
    capacity, including residual
140         These edges count as valid paths to
    neighbouring nodes in Edmonds-Karp"""
141         graph = self.graph
142         a, b, c, d, *_ = graph.nodes()
143         graph.add_edge(a, (b, 10))
144         graph.add_edge(b, (c, 2))
145         graph.add_edge(c, (d, 10))
146         graph.add_edge(d, (a, 10))
147 
148         graph.to_dot()
149 
150         MaxFlow.augment_flow(graph, [a, b, c, d], 2
    )
151         c_flow_neighbours = graph.flow_neighbours(c
    )
152         self.assertEqual([d, b], GenericDigraph.
    nodes_from_edges(c_flow_neighbours))
153 
154     def test_bool(self):
155         """Tests that dunder bool method returns
    true when there are edges in the graph"""
156         # empty
157         print(self.graph.graph)
158         self.assertFalse(bool(self.graph))
159 
160         # with an edge
161         a, b, c, d, e = self.nodes
162         self.graph.add_edge(a, (b, 10))
163 
164     def test_add_existing(self):
165         """Tests that adding an existing edge will
    add capacity onto the existing edge,
166         as opposed to adding a new distinct edge
    from src -> dest"""
167         a, b, c, d, e = self.graph.nodes()
168         self.graph.add_edge(a, (b, 5))
169         self.graph.add_edge(a, (b, 5))
170         print(self.graph)
```

```
171         self.assertEqual(self.graph.get_edge(a, b).
    capacity, 10)
172 
173     def test_pop_existing(self):
174         """Tests that if an edge A -> B of weight [
    5] exists, adding B -> A [5] will remove
175         any edges between A and B, as there is net
    0 between the two"""
176         a, b, c, d, e = self.graph.nodes()
177         self.graph.add_edge(a, (b, 5))
178         self.graph.add_edge(b, (a, 5))
179 
180         self.assertFalse(self.graph.is_edge(a, b))
181 
182     def test_add_edge_reverses_new_edge(self):
183         """Checks that if there exists an edge A
     -> B of weight [5] exists, adding B -> A [10] will
184         reverse the direction of the edge such that
     now an edge B -> A [5] exists"""
185         a, b, c, d, e = self.graph.nodes()
186         self.graph.add_edge(a, (b, 5))
187         self.graph.add_edge(b, (a, 10))
188 
189         self.assertFalse(self.graph.is_edge(a, b))
190         self.assertEqual(self.graph.get_edge(b, a).
    capacity, 5)
191 
192 
193 class TestMaxFlow(TestCase):
194     def setUp(self) -> None:
195         graph = FlowGraph([Vertex(n, label=chr(n +
    97)) for n in range(4)])
196         a, b, c, d, *_ = graph.nodes()
197         graph.add_edge(a, (b, 10))
198         graph.add_edge(b, (c, 2))
199         graph.add_edge(c, (d, 10))
200         graph.add_edge(d, (a, 10))
201 
202         self.graph = graph
203 
204     def test_bottleneck(self):
```

```
205         """Checks we get the correct bottleneck
    value on a path"""
206         # build a chain with an obvious bottleneck
    on normal graphs
207         graph = self.graph
208         a, b, c, d, *_ = graph.nodes()
209 
210         aug_path = [a, b, c, d]
211 
212         graph.to_dot()
213 
214         self.assertEqual(2, MaxFlow.bottleneck(
    graph, aug_path))
215 
216     def test_nodes_to_path(self):
217         """Checks that we can build a path of edges
     from nodes"""
218         a, b, c, d, *_ = self.graph.nodes()
219         edges = MaxFlow.nodes_to_path(self.graph, [
    a, b, c, d])
220 
221         self.assertEqual(
222             edges, [FlowEdge(a, b, 10), FlowEdge(b
    , c, 2), FlowEdge(c, d, 10)]
223         )
224 
225     def test_augmenting_path(self):
226         """Tests that we find valid augmenting
    paths"""
227         a, b, c, d, *_ = self.graph.nodes()
228         MaxFlow.augmenting_path(self.graph, a, d)
229 
230     def test_augment_flow(self):
231         """Checks that augmenting flow works
    exactly as described in Analysis"""
232         a, b, c, d, *_ = self.graph.nodes()
233 
234         MaxFlow.augment_flow(self.graph, [a, b, c,
    d], 2)
235         self.graph.to_dot()
236 
```

```
237         # check that the edges and residual edges
    all now have flow 2
238         node_path = [a, b, c, d]
239         flow = 2
240         for _ in range(2):
241             # one for normal one for residual
242             for src, dest in zip(node_path,
    node_path[1:]):
243                 with self.subTest(f"{src} -> {dest}
    "):
244                     self.assertEqual(flow, self.
    graph.get_edge(src, dest).flow)
245             # change path to be residual, flow
    changes accordingly
246             node_path.reverse()
247             flow *= -1
248 
249     def test_edmonds_karp(self):
250         """Integration test for edmonds-karp,
    ensures that max flow is correct"""
251         # build slightly more involved graph
252         nodes = [Vertex(0, label="src"), Vertex(10
    , label="sink")]
253         nodes += [Vertex(n, label=chr(n + 96)) for
    n in range(1, 10)]
254         tg = FlowGraph(nodes)
255 
256         s, t, a, b, c, d, e, f, g, h, i = tg.nodes
    ()
257 
258         tg.add_edge(s, (a, 5), (b, 10), (c, 5))
259         tg.add_edge(a, (d, 10))
260         tg.add_edge(b, (a, 15), (e, 20))
261         tg.add_edge(c, (f, 10))
262         tg.add_edge(d, (e, 25), (g, 10))
263         tg.add_edge(e, (c, 5), (h, 30))
264         tg.add_edge(f, (h, 5), (i, 5))
265         tg.add_edge(g, (t, 5))
266         tg.add_edge(h, (t, 15), (i, 5))
267         tg.add_edge(i, (t, 10))
268 
```

```
269         tg.to_dot()
270 
271         max_flow = MaxFlow.edmonds_karp(tg, s, t)
272         self.assertEqual(max_flow, 20)
273 
274     def test_old_edmonds(self):
275         """Tests a more complex graph for correct
    maxflow"""
276         labels = ["a", "b", "c", "d", "e", "f"]
277         self.vertices = [Vertex(ID, label=label)
    for ID, label in enumerate(labels)]
278         a, b, c, d, e, f = self.vertices
279 
280         # set up weighted_graph and flow graphs
281         self.flow_graph = FlowGraph(self.vertices)
282         self.flow_graph.add_edge(a, (b, 10), (c, 10
    ))
283         self.flow_graph.add_edge(b, (d, 25))
284         self.flow_graph.add_edge(c, (e, 25))
285         self.flow_graph.add_edge(d, (f, 10))
286         self.flow_graph.add_edge(e, (f, 10), (b, 6
    ))
287 
288         a, b, c, d, e, f = self.vertices
289 
290         expected = 20
291         calculated = MaxFlow.edmonds_karp(self.
    flow_graph, a, f)
292 
293         self.assertEqual(expected, calculated)
294 
295 
296 class TestSimplify(TestCase):  # type: ignore
297     def setUp(self) -> None:
298         self.graph = FlowGraph([Vertex(0, "d"),
    Vertex(1, "m"), Vertex(2, "t")])
299         d, m, t = self.graph.nodes()
300         self.graph.add_edge(d, (m, 5), (t, 10))
301         self.graph.add_edge(m, (t, 5))
302 
303     def test_simplify_debt(self):
```

```
304         """Uses graph from analysis to test that
    the debt simplifying algorithm simplifies debt,
    does not create or
305         destroy debt"""
306
307         print(self.graph.net_debt)
308
309         people = ["dad", "tom", "maia"]
310         debt = FlowGraph([Vertex(ID, person) for ID
    , person in enumerate(people)])
311
312         d, t, m = debt.nodes()
313
314         debt.add_edge(d, (t, 10), (m, 5))
315         debt.add_edge(m, (t, 5))
316
317         debt.to_dot()
318
319         clean = Simplify.simplify_debt(debt)
320
321         clean.to_dot(n=1)
322         self.assertEqual(debt.net_debt, clean.
    net_debt)
323
324     def test_adjust_edges(self):
325
326         """Checks that edges are adjusted
    accordingly, generate graphs before and after"""
327
328         # saturate d -> m -> t
329         self.graph.to_dot(n=0)
330         d, m, t = self.graph.nodes()
331         MaxFlow.augment_flow(self.graph, [d, m, t
    ], 5)
332         self.graph.to_dot(n=1)
333         self.graph.adjust_edges()
334         self.graph.to_dot(n=4)
335
336         # graph should have no edges in or out of m
337         # t should have a res edge to d, 0/0
338         # d should have a fwd to t, 0/10
```

```
339
340         self.assertFalse(self.graph[m])
341         self.assertEqual(self.graph[d], [FlowEdge(d
    , t, 10)])
342         self.assertEqual(self.graph[t], [FlowEdge(t
    , d, 0)])
343
344     def test_mithun_simplify(self):
345
346         """Checks more complex example, ensures
    that graph is settled and that no debt is created
     / destroyed"""
347
348         # gen vertices
349         people: list[Vertex] = []
350         for ID, person in enumerate(["b", "c", "d"
    , "e", "f", "g"]):
351             people.append(Vertex(ID, label=person))
352         b, c, d, e, f, g = people
353
354         # build flow graph of transactions
355         messy = FlowGraph(people)
356
357         messy.add_edge(b, (c, 40))
358         messy.add_edge(c, (d, 20))
359         messy.add_edge(d, (e, 50))
360         messy.add_edge(f, (e, 10), (d, 10), (c, 30
    ), (b, 10))
361         messy.add_edge(g, (b, 30), (d, 10))
362
363         messy.to_dot()
364
365         clean = Simplify.simplify_debt(messy)
366         clean.to_dot(n=1)
367
368         self.assertEqual(messy.net_debt, clean.
    net_debt)
369
370     def test_simplest_graph(self):
371         """Shouldn't change"""
372         people = ["dad", "tom", "maia"]
```

```
373         debt = FlowGraph([Vertex(ID, person) for ID
    , person in enumerate(people)])
374         d, t, m = debt.nodes()
375         debt.add_edge(d, (t, 5))
376         debt.add_edge(t, (m, 10))
377 
378         debt.to_dot()
379 
380         with self.assertRaises(NoOptimisations):
381             clean = Simplify.simplify_debt(debt)
382             clean.to_dot(n=1)
383 
```

File - /home/tcassar/projects/settle/tests/test_settling/test_graph.py

```python
# coding=utf-8

from unittest import TestCase

from src.simplify.base_graph import Digraph
from src.simplify.flow_graph import *
from src.simplify.graph_objects import Edge
from src.simplify.weighted_digraph import
WeightedDigraph


class TestDigraph(TestCase):
    """Generate a digraph and some vertices"""

    def setUp(self) -> None:
        """Build basic graph"""
        labels = ["u", "v", "w"]
        self.vertices = [Vertex(ID, label=label) for
 ID, label in enumerate(labels)]

        self.graph = Digraph(self.vertices)
        u, v, w = self.vertices
        self.graph.add_edge(u, v, w)
        self.graph.add_edge(v, w)

    def test_init(self):
        expected = "U -> VW\nV -> W\nW -> \n"

        with self.subTest("init"):
            self.assertEqual(expected, str(self.
graph))

        with self.subTest("helpers"):
            self.assertTrue(self.graph.is_node(self.
vertices[0]))
            self.assertTrue(self.graph.sanitize(*
self.vertices))

            with self.assertRaises(GraphError):
                self.graph.edge_from_nodes(self.
vertices[2], [Edge(self.vertices[0])])
```

```
36
37      def test_is_edge(self):
38          u, v, w = self.vertices
39
40          with self.subTest("is edge"):
41              self.assertTrue(self.graph.is_edge(u, v
    ))
42          with self.subTest("isn't edge"):
43              self.assertFalse(self.graph.is_edge(w, v
    ))
44
45      def test_add_node(self):
46
47          new = Vertex(4, "b")
48          self.assertFalse(self.graph.is_node(new))
49
50          self.graph.add_node(new)
51          self.assertTrue(self.graph.is_node(new))
52
53      def test_pop_node(self):
54          u, v, w = self.vertices
55
56          print(self.graph)
57          self.graph.pop_node(u)
58          print(self.graph)
59          self.assertFalse(self.graph.is_node(u))
60          with self.assertRaises(GraphError):
61              self.graph.edge_from_nodes(u, self.graph
    [v])
62
63      def test_pop_edge(self):
64          u, v, w = self.vertices
65          self.assertTrue(self.graph.is_edge(u, v))
66          self.graph.pop_edge(u, v)
67
68          self.assertFalse(self.graph.is_edge(u, v))
69
70      def test_add_edge(self):
71          u, v, w = self.graph.nodes()
72          self.assertFalse(self.graph.is_edge(w, v))
73          self.graph.add_edge(w, v)
```

```
 74
 75         self.assertTrue(self.graph.is_edge(w, v))
 76
 77     def test_nodes(self):
 78         self.assertEqual(self.vertices, self.graph.
    nodes())
 79
 80
 81 class TestWeightedDigraph(TestCase):
 82     """Artefact of prototyping process"""
 83     def setUp(self) -> None:
 84         """Build basic graph"""
 85         labels = ["u", "v", "w"]
 86         self.vertices = [Vertex(ID, label=label)
    for ID, label in enumerate(labels)]
 87
 88         self.graph = WeightedDigraph(self.vertices)
 89         u, v, w = self.vertices
 90         self.graph.add_edge(u, (v, 1), (w, 2))
 91         self.graph.add_edge(v, (w, 3))
 92
 93     def test_add_edge(self):
 94         u, v, w = self.vertices
 95         self.assertFalse(self.graph.is_edge(w, v))
 96         self.graph.add_edge(w, (v, 4))
 97         self.assertTrue(self.graph.is_edge(w, v))
 98
 99     def test_add_existing_edge(self):
100         u, v, w = self.graph.nodes()
101         self.assertEqual(self.graph.is_edge(v, w),
    3)
102         self.graph.add_edge(v, (w, 2))
103         self.assertEqual(self.graph.is_edge(v, w),
    5)
104
105     def test_flow_through(self):
106         for node, flow in zip(self.vertices, [3, 2
    , -5]):
107             with self.subTest(node):
108                 self.assertEqual(self.graph.
    flow_through(node), flow)
```

```
1 ID,src,dest,amount,group,src_n,src_e,dest_n,dest_e,
  src_d,dest_d
2 0,04,20,5,0,
  216167920311437527463094155794523202025108931260730876523837234081050636000701477615009364545704708627869702069833686361660030089751732511247633740543213460303544570214251653560377816369300361064044182954137264310025568369000670498679444999043128421557451025437279819137427553642625019821057158620227234339857381390676943784764665144556770107968670906167894855774586373708692617743377046549318417755362244202417503903181827421441408785602795322818434551136750209831842525036691384924518828835962027759009110005026961294337702970618450353224231935687225848095890069620605874219546032017844978461582635821441774306426034,65537,
  24543526053222685920089002431387071080808753979168289339650707414109582770834242173009539282426745682722988055346700400544867910943785143909523891526583685873084778519448472887002660212182814126070863004440351914152540555901097499930235793743416545610368040032596055789846809140241500787296582367699342532180447446042851559187629052238168999819843726156363830030069139123051200441111945407276884741033640730462395861985143994176991609920694746571704707094532698910844357727419783667461542340930287506515793569704149384784131969168222943901917462927710491717935881329934336642631167278369058626398091411958822782486021,65537,
  42318605026103475553526985044230475639442568138229963925734038989576569874864579669508371562955146126547877955291914430323301128393346172685767442683818738967216638476720764074944647817540070989695696588210951082558372250397158894945714319204774976709066583095965638944919595239862657801000401988151966317962866310493446444221800190039054548479140813630569444720847689992964879750244948034517871603890465032993904165597478235001604038074703682680413388703757501362444680122222701317562224831854480644662034354542275281227314387046825525570376824278684460211954070416261823570911887469407283866423584255451231566190517451642798421916838978706887752529294378563794943959644593481018317996812806135240585387347011403463919
```

```
2
484312360288671519765089185961940677538082993405248358724852816062664396208614460346356077862920426734827642392974806673957359914994399933521690425914379137265608737153994017658024271602617427327911897090105542806007602652290786721234528626131896832489562551558690301697229495263298457688434655931885453437842161128989953202272867367837891847562410504381902662947168208004596529501772278874463251649170353674520402106924748077589683971300268994119580437205561936946700600893560778543423518093933835790069197852673
3 1,04,13,10,0,
216167920311437527463094155794523202025108931260730876523837234081050636000701477615009364545704708627869702069833686361660030089751732511247633740543213460303544570214251653560377816369300361064044182954137264310025568369000670498679444999043128421557451025437279819137427553642625019821057158620227234339857381390676943784764665144556770107968670906167894855774586373708692617743377046549318417755362244202417503903181827421441408785602795322818434551136750209831842525036691384924518828835962027759009110005026961294337702970618450353224231935687225848095890069620605874219546032017844978461582635821441774306426 03,65537,
225658640372474598320855668799003073602163686974087338839648518084108748208582879216599979547434442126258889014725245633691567258383521676090489827618604882344659937010912157571801294412422946171359122243583308263282848559863033353276205119720064531420906643446610551234084941258572108217677529623697862636846419040961276021458997843951217963241478789474394699527160580771635153728361835033613801478959059921454822591898567781991786261932874106056739293869626830428217313621069552445747883424421554798845480580652219017109280678813179596281553889038339230703284433283787263864958666155092517689554259271552956245888 23,65537,
42318605026103475553526985044230475639442568138229963925734038989576569874864579669508371562955146126547877955291914430323301128393346172685767442683818738967216638476720764074944647817540070989695696588210951082558372250397158894945714319204774976709066583095
```

```
3
   965638944919595239862657801000401988151966317962866310
   493446644422180019003905454847914081363056944472084768
   999296487975024494803451787160389046503299390416559747
   823500160403807470368268041338870375750136244468012222
   701317562224831854480644662034354542275281227314387046
   825525570376824278684460211954070416261823570911887469
   407283866423584255451231566190577541428218992751486727
   644862498271472919514635342583131191306090515724089556
   837205820094594019006617381787091438615311514626770508
   215330966596440453041517774718735698700608641179259809
   996868147443542318468896990846846049836207197701758385
   280007526246224538969096382406756380016525685903567419
   429111431200416762356947622267619084456004720872031090
   989474470600876826405729239073983735694492507804544435
   298169779944004811276378982537638262601152485343855539
   104875876907807892323552073225968951530325990045227310
   213319704081395654861368829855439175850787537877943284
   782407352583525021787084903098686611464144261278387324
   2,20,13,5,0,24543526053222685920089002431387071080808
   753979168289339650707414109582770834242173009539282426
   745682722988055346700400544867910943785143909523891526
   583685873084778519448472887002660212182814126070863004
   440351914152540555901097499930235793743416545610368040
   032596055789846809140241500787296582367699342532180447
   446042851559187629052238168999819843726156363830030069
   139123051200441111945407276884741033640730462395861985
   143994176991609920694746571704707094532698910844357727
   419783667461542340930287506515793569704149384784131969
   168222943901917462927710491717935881329934336642631167
   278369058626398091411958822782486021,65537,225658640372
   474598320855668799003073602163686974087338839648518084
   108748208582879216599979547434442126258889014725245633
   691567258383521676090489827618604882344659937010912157
   571801294412422946171359122243583308263282848559863033
   353276205119720064531420906643446610551234084941258572
   108217677529623697862636846419040961276021458997843951
   217963241478789474394699527160580771635153728361835033
   613801478959059921454822599
```

```
4
  189856778199178626193287410605673929386962683042821731362106955244574788342442155479884548058065221901710928067881317959628155388903833923070328443328378726386495866615509251768955425927155295624588823,65537,1745164279842191683897870688775252929437856379494395964459348101831799681280613524058538734701140346391948431236028867151976508918596194067753808299340524835872485281606266439620861446034635607786292042673482764239297480667395735991499439993352169042591437913726560873715399401765802427160261742732791189709010554280600760265229078672123452862613189683248956255155869030169722949526329845768843465593188545343784216112898995320227286736783789184756241050438190266294716820800459652950177227887446325164917035367452040210692474807758968397130026899411958043720556193694670060089356077854342351809393383579006919785267360077543428218992751486727644862498271472919514635342583131191306090515724089556837205820094594019006617381787091438615311514626770508215330966596440453041517774718735698700608641179259809996868147443542318468896990846846049836207197701758385280007526246224538969096382406756380016525685903567419429111431200416762356947622267619084456004720872031090989474470600876826405729239073983735694492507804544435298169779944004811276378982537638262601152485343855539104875876907807892323552073225968951530325990045227310213319704081395654861368829855439175850787537877943284782407352583525021787084903098686611464144261278387358352502178708490309868661146414426127838735 3,04,20,5,1,2161679203114375274630941557945232020251089312607308765238372340810506360007014776150093645457047086278697020698336863616600300897517325112476337405432134603035445702142516535603778163693003610640441829541372643100255683690006704986794449990431284215574510254372798191374275536426250198210571586202272343398573813906769437847646651445567701079686709061678948557745863737086926177433770465493184177553622442024175039031818274214414087856027953228184345511367502098318425250366913849245188288359620277590091100050269612943377029706184503532242319356872258480958900696206058 7
```

```
5 42195460320178449784615826358214417743064260 3,65537,
  24543526053222685920089002431387071080808753979168289
  33396507074141095827708342421730095392824267456827229
  88055346700400544867910943785143909523891526583685873
  08477851944847288700266021218281412607086300444035191
  41525405559010974999302357937434165456103680400325960
  55789846809140241500787296582367699342532180447446042
  85155918762905223816899981984372615636383003006913912
  30512004411119454072768847410336407304623958619851439
  94176991609920694746571704707094532698910844357727419
  78366746154234093028750651579356970414938478413196916
  82229439019174629277104917179358813299343366426311672
  78369058626398091411958822782486021,65537,42318605026
  10347555352698504423047563944256813822996392573403898
  95765698748645796695083715629551461265478779552919144
  30323301128393346172685767442683818738967216638476720
  76407494464781754007098969569658821095108255837225039
  71588949457143192047749767090665830959656389449195952
  39862657801000401988151966317962866310493446444221800
  19003905454847914081363056944472084768999296487975024
  49480345178716038904650329939041655974782350016040380
  74703682680413388703757501362444680122222701317562224
  83185448064466203435454227528122731438704682552557037
  68242786844602119540704162618235709118874694072838664
  23584255451231566190 5,
  17451642798421916838978706887752529294378563794943959
  64459348101831799681280613524058538734701140346391948
  43123602886715197650891859619406775380829934052483587
  24852816062664396208614460346356077862920426734827642
  39297480667395735991499439993352169042591437913726560
  87371539940176580242716026174273279118970901055428060
  07602652290786721234528626131896832489562551558690301
  69722949526329845768843465593188545343784216112898995
  32022728673678378918475624105043819026629471682080045
  96529501772278874463251649170353674520402106924748077
  58968397130026899411958043720556193694670060089356077
  85434235180939338357900691978526736 4,04,13,10,1,
  21616792031143752746309415579452320202510893126073087
  65238372340810506360007014776150093645457047086278697
  02069833686361660030089751732511247633740543213460
```

```
6
30354457021425165356037781636930036106404418295413726
64310025568369000670498679444999043128421557451025 43
72798191374275536426250198210571586202272343398573 81
39067694378476466514455677010796867090616789485577 45
86373708692617743377046549318417755362244202417503 90
31818274214414087856027953228184345511367502098318 42
52503669138492451882883596202775900911000502696129 43
37702970618450353224231935687225848095890069620605 87
42195460320178449784615826358214417743064260 3,65537,
22565864037247459832085566879900307360216368697408 73
38839648518084108748208582879216599979547434442126 25
88890147252456336915672583835216760904898276186048 82
34465993701091215757180129441242294617135912224358 33
08263282848559863033353276205119720064531420906643 44
66105512340849412585721082176775296236978626368464 19
04096127602145899784395121796324147878947439469952 71
60580771635153728361835033613801478959059921454822 59
18985677819917862619328741060567392938696268304282 17
31362106955244574788342442155479884548058065221901 71
09280678813179596281553889038339230703284433283787 26
38649586661550925176895542592715529562458882 3,65537,
42318605026103475553526985044230475639442568138229 96
39257340389895765698748645796695083715629551461265 47
87795529191443032330112839334617268576744268381873 89
67216638476720764074944647817540070989695696588210 95
10825583722503971588949457143192047749767090665830 95
96563894491959523986265780100040198815196631796286 63
10493446444221800190039054548479140813630569444720 84
76899929648797502449480345178716038904650329939041 65
59747823500160403807470368268041338870375750136244 46
80122222701317562224831854480644662034354542275281 22
73143870468255255703768242786844602119540704162618 23
57091188746940728386642358425545123156619 05,
77541428218992751486727644862498271472919514635342 58
31311913060905157240895568372058200945940190066173 81
78709143861531151462677050821533096659644045304151 77
74718735698700608641179259809996868147443542318468 89
69908468460498362071977017583852800075262462245389 69
09638240675638001652568590356741942911143120041676 23
56947622267619084456004720872031090989474470600876 82
```

```
6
64057292390739837356944925078045444352981697799440048112763789825376382626011524853438555391048758769078078923235520732259689515303259900452273102133197040813956548613688298554391758507875378779432847824073525835250217870849030986866114641442612783873
7 5,20,13,5,1,
24543526053222685920089002431387071080808753979168289339650707414109582770834242173009539282426745682722988055346700400544867910943785143909523891526583685873084778519448472887002660212182814126070863004440351914152540555901097499930235793743416545610368040032596055789846809140241500787296582367699342532180447446042851559187629052238168999819843726156363830030069139123051200441111945407276884741033640730462395861985143994176991609920694746571704707094532698910844357727419783667461542340930287506515793569704149384784131969168222943901917462927710491717935881329934336642631167278369058626398091411958822782486021,65537,225658640372474598320855668799003073602163686974087338839648518084108748208582879216599979547434442126258889014725245633691567258383521676090489827618604882344659937010912157571801294412422946171359122243583308263282848559863033353276205119720064531420906643446610551234084941258572108217677529623697862636846419040961276021458997843951217963241478789474394699527160580771635153728361835033613801478959059921454822591898567781991786261932874106056739293869626830428217313621069552445747883424421554798845480580652219017109280678813179596281553889038339230703284433283787263864958666155092517689554259271552956245888 23,65537,17451642798421916838978706887752529294378563794943959644593481018317996812806135240585387347011403463919484312360288671519765089185961940677538082993405248358724852816062664396208614460346356077862920426734827642392974806673957359914994399933521690425914379137265608737153994017658024271602617427327911897090105542806007602652290786721234528626131896832489562551558690301697229495263298457688434655931885453437842161128989953202272867367837891847562410504381902662947168208004596529501772278874463251649170353674520402 10
```

```
7
  6924748077589683971300268994119580437205561936946700
  6008935607785434235180939338357900691978526 73,
  7754142821899275148672764486249827147291951463534258
  3131191306090515724089556837205820094594019006617381
  7870914386153115146267705082153309665964404530415177
  7471873569870060864117925980999686814744354231846889
  6990846846049836207197701758385280007526246224538969
  0963824067563800165256859035674194291114312004167623
  5694762226761908445600472087203109098947447060087682
  6405729239073983735694492507804544435298169779944004
  8112763789825376382626011524853438555391048758769078
  0789232355207322596895153032599004522731021331970408
  1395654861368829855439175850787537877943284782407352
  5835250217870849030986866114641442612783873
8 6,04,20,5,2,
  2161679203114375274630941557945232020251089312607308
  7652383723408105063600070147761500936454570470862786
  9702069833686361660030089751732511247633740543213460
  3035445702142516535603778163693003610640441829541372
  6431002556836900067049867944499043128421557451025 43
  7279819137427553642625019821057158620227234339857381
  3906769437847646651445567701079686709061678948557745
  8637370869261774337704654931841775536224420241750390
  3181827421441408785602795322818434551136750209831842
  5250366913849245188288359620277590091100050269612943
  3770297061845035322423193568722584809589006962060587
  4219546032017844978461582635821441774306426 03,65537,
  2454352605322268592008900243138707108080875397916828
  9339650707414109582770834242173009539282426745682722
  9880553467004005448679109437851439095238915265836858
  7308477851944847288700266021218281412607086300444035
  1914152540555901097499930235793743416545610368040032
  5960557898468091402415007872965823676993425321804474
  4604285155918762905223816899981984372615636383003006
  9139123051200441111945407276884741033640730462395861
  9851439941769916099206947465717047070945326989108443
  5772741978366746154234093028750651579356970414938478
  4131969168222943901917462927710491717935881329934336
  6426311672783690586263980914119588227824860 21,65537,
  4231860502610347555352698504423047563944256813822996
```

```
8
   392573403898957656987486457966950837156295514612654787795529191443032330112839334617268576744268381873896721663847672076407494464781754007098969569658821095108255837225039715889494571431920477497670906658309596563894491959523986265780100040198815196631796286631049344644422180019003905454847914081363056944472084768999296487975024494803451787160389046503299390416559747823500160403807470368268041338870375750136244468012222701317562224831854480644662034354542275281227314387046825525570376824278684460211954070416261823570911887469407283866423584255451231566190517451642798421916838978706887752529294378563794943959644593481018317996812806135240585387347011403463919484312360288671519765089185961940677538082993405248358724852816062664396208614460346356077862920426734827642392974806673957359914994399933521690425914379137265608737153994017658024271602617427327911897090105542806007602652290786721234528626131896832489562551558690301697229495263298457688434655931885453437842161128989953202272867367837891847562410504381902662947168208004596529501772278874463251649170353674520402106924748077589683971300268994119580437205561936946700600893560778543423518093933835790069197852673
9 7,04,13,10,2,65537,
   621616792031143752746309415579452320202510893126073087652383723408105063600070147761500936454570470862786970206983368636166003008975173251124763374054321346030354457021425165356037781636930036106404418295413726431002556836900067049867944499043128421557451025437279819137427553642625019821057158620227234339857381390676943784764665144556770107968670906167894855774586373708692617743377046549318417755362244202417503903181827421441408785602795322818434551136750209831842525036691384924518828835962027759009110005026961294337702970618450353224231935687225848095890069620605874219546032017844978461582635821441774306426035537,225658640372474598320855668799003073602163686974087338839648518084108748208582879216599979547434442126258889014725245633691567258383521676090489827618604882344659937010912157571801294412422946171359122243583
```

```
 9
   0826328284855986303335327620511972006453142090664344
   6610551234084941258572108217677529623697862636846419
   0409612760214589978439512179632414787894743946995271
   6058077163515372836183503361380147895905992145482259
   1898567781991786261932874106056739293869626830428217
   3136210695524457478834244215547988454805806522190171
   0928067881317959628155388903833923070328443328378726
   3864958666155092517689554259271552956245888823,65537,
   4231860502610347555352698504423047563944256813822996
   3925734038989576569874864579669508371562955146126547
   8779552919144303233011283933461726857674426838187389
   6721663847672076407494464781754007098969569658821095
   1082558372250397158894945714319204774976709066583095
   9656389449195952398626578010004019881519663179628663
   1049344644422180019003905454847914081363056944472084
   7689992964879750244948034517871603890465032993904165
   5974782350016040380747036826804133887037575013624446
   8012222270131756222483185448064466203435454227528122
   7314387046825525570376824278684460211954070416261823
   5709118874694072838664235842554512315661905,
   7754142821899275148672764486249827147291951463534258
   3131191306090515724089556837205820094594019006617381
   7870914386153115146267705082153309665964404530415177
   7471873569870060864117925980999686814744354231846889
   6990846846049836207197701758385280007526246224538969
   0963824067563800165256859035674194291114312004167623
   5694762226761908445600472087203109098947447060087682
   6405729239073983735694492507804544435298169779944004
   8112763789825376382626011524853438555391048758769078
   0789232355207322596895153032599004522731021331970408
   1395654861368829855439175850787537877943284782407352
   5835250217870849030986866114641442612783873
10 8,20,13,5,2,
   2454352605322268592008900243138707108080875397916828
   9339650707414109582770834242173009539282426745682722
   9880553467004005448679109437851439095238915265836858
   7308477851944847288700266021218281412607086300444035
   1914152540555901097499930235793743416545610368040032
   5960557898468091402415007872965823676993425321804474
   4604285155918762905223816899981984372615636383003006
```

```
10
    913912305120044111194540727688474103364073046239586 1
    985143994176991609920694746571704707094532698910844 3
    577274197836674615423409302875065157935697041493847 8
    413196916822294390191746292771049171793588132993433 6
    642631167278369058626398091411958822782486021,65537,
    225658640372474598320855668799003073602163686974087 3
    388396485180841087482085828792165999795474344421262 5
    888901472524563369156725838352167609048982761860488 2
    344659937010912157571801294412422946171359122243583 3
    082632828485598630333532762051197200645314209066434 4
    661055123408494125857210821767752962369786263684641 9
    040961276021458997843951217963241478789474394699527 1
    605807716351537283618350336138014789590599214548225 9
    189856778199178626193287410605673929386962683042821 7
    313621069552445747883424421554798845480580652219017 1
    092806788131795962815538890383392307032844332837872 6
    386495866615509251768955425927155295624588823,65537,
    174516427984219168389787068877525292943785637949439 5
    964459348101831799681280613524058538734701140346391 9
    484312360288671519765089185961940677538082993405248 3
    587248528160626643962086144603463560778629204267348 2
    764239297480667395735991499439993352169042591437913 7
    265608737153994017658024271602617427327911897090105 5
    428060076026522907867212345286261318968324895625515 5
    869030169722949526329845768843465593188545343784216 1
    128989953202272867367837891847562410504381902662947 1
    682080045965295017722788744632516491703536745204021 0
    692474807758968397130026899411958043720556193694670 0
    600893560778543423518093933835790069197852673,
    775414282189927514867276448624982714729195146353425 8
    313119130609051572408955683720582009459401900661738 1
    787091438615311514626770508215330966596440453041517 7
    747187356987006086411792598099968681474435423184688 9
    699084684604983620719770175838528000752624622453896 9
    096382406756380016525685903567419429111431200416762 3
    569476222676190844560047208720310909894744706008768 2
    640572923907398373569449250780454443529816977994400 4
    811276378982537638262601152485343855539104875876907 8
    078923235520732259689515303259900452273102133197040 8
    139565486136882985543917585078753787794328478240735 2
```

```
10 583525021787084903098686611464144261278387 3
11
```

```python
# coding=utf-8
import unittest

import src.simplify.flow_graph
import src.simplify.graph_objects
from src.crypto import keys
from src.transactions.ledger import *
from src.transactions.transaction import VerificationError


def setUpModule():
    print(os.getcwd())
    os.chdir("/home/tcassar/projects/settle")


def key_path(usr: str, keytype="private") -> str:
    assert usr == "d" or usr == "m" or usr == "t"
    return f"./src/crypto/sample_keys/{usr}_{keytype}-key.pe{'m' if keytype == 'private' else ''}"


class TestLedger(unittest.TestCase):
    def setUp(self) -> None:

        """Load in sample data from mock db"""

        self.valid: Ledger
        self.missing_key: Ledger
        self.invalid: Ledger

        # load in raw copies of d, m, t test keys
        ldr = keys.RSAKeyLoader()
        self.d_m_t_keys: dict[int, keys.RSAPrivateKey] = {}
        pubs: list[keys.RSAPublicKey] = []

        for person, id in zip(["d", "m", "t"], [4, 13, 20]):
            ldr.load(key_path(person))
            ldr.parse()
```

```
38              self.d_m_t_keys[id] = keys.RSAPrivateKey
   (ldr)
39              pubs.append(keys.RSAPublicKey(ldr))
40
41          self.d_pub, self.m_pub, self.t_pub = pubs
42          self.d_priv, self.m_priv, self.t_priv = self
   .d_m_t_keys.values()
43
44          self.valid, self.missing_key, self.invalid
    = LedgerLoader.load_from_csv(
45              "./tests/test_transactions/mock_db.csv"
46          )
47
48      def test_add(self):
49          """test adding transactions to a ledger"""
50
51          with self.subTest("Add"):
52              ledger = Ledger()
53              self.assertFalse(not not ledger)
54              ledger.append(Transaction(0, 0, 0, self.
   d_priv, self.m_priv))
55              self.assertTrue(not not ledger)
56
57          with self.subTest("Catch non transaction"),
   self.assertRaises(LedgerBuildError):
58              ledger.append(6)  # type: ignore
59
60      def test_load_from_csv(self):
61          """Test loading from mock db works as
   expected"""
62          ledger_list = LedgerLoader.load_from_csv(
63              "./tests/test_transactions/mock_db.csv"
64          )
65          self.assertEqual(len(ledger_list), 3)
66
67      def test_verify_transactions(self):
68          """
69          Make three ledgers:
70              1) Valid transactions
71              2) An unsigned transaction
72              3) An invalid signature
```

```
 73
 74         Check that first one goes through no issues
    , and that other two are caught
 75         """
 76
 77         with self.subTest("unsigned"), self.
    assertRaises(VerificationError):
 78             self.valid._verify_transactions()
 79
 80         # sign transactions in ledger; not normal
    procedure so contrived
 81         self.sign()
 82
 83         with self.subTest("signed"):
 84             self.valid._verify_transactions()
 85
 86         with self.subTest("missing key"), self.
    assertRaises(VerificationError):
 87             self.missing_key._verify_transactions()
 88
 89         with self.subTest("invalid key"), self.
    assertRaises(VerificationError):
 90             self.invalid._verify_transactions()
 91
 92     def test_as_flow(self):
 93         """Test that we build flow graphs from
    ledgers as expected"""
 94         # sign ledger
 95         self.sign()
 96
 97         Vertex = src.simplify.graph_objects.Vertex
 98
 99         exp = src.simplify.flow_graph.FlowGraph(
100             [Vertex(ID=4), Vertex(ID=13), Vertex(ID
    =20)]
101         )
102         d, m, t = exp.nodes()
103         exp.add_edge(d, (m, 10), (t, 5))
104         exp.add_edge(t, (m, 5))
105
106         as_flow = self.valid._as_flow()
```

```
107
108         with self.subTest("nodes"):
109             self.assertEqual(exp.nodes(), self.
    valid.nodes)
110
111         with self.subTest("to flow graph"):
112             self.assertEqual(exp, as_flow)
113
114     def sign(self):
115         """Checks that signing a ledger is a
    reliable process"""
116         for trn in self.valid.ledger:
117             trn.sign(self.d_m_t_keys[trn.src],
    origin="src")
118             trn.sign(self.d_m_t_keys[trn.dest],
    origin="dest")
119             print("verifying...")
120             trn.verify()
121             print("verified")
122
123     def test_flow_to_transactions(self):
124         """Checks that flow graph data structures
    can be effectively converted to ledgers"""
125         self.maxDiff = None
126         self.sign()
127         as_flow = self.valid._as_flow()
128
129         # remove sigs, ID, for comparison
130         trn: Transaction
131         for trn in self.valid.ledger:
132             trn.ID = 0
133             trn.signatures = {}
134
135         with self.subTest("to transactions"):
136             self.valid.ledger.sort(key=lambda trn:
    trn.amount)
137             calc: list[Transaction] = self.valid.
    _flow_to_transactions(as_flow)
138             calc.sort(key=lambda trn: trn.amount)
139
140             self.assertEqual(calc, self.valid.
```

```
140 ledger)
141
142     def test_simplify_ledger(self):
143         """Integration test, checks that calling
    simplify() on a ledger displays expected behaviour
    """
144         self.sign()
145         self.valid.simplify_ledger()
146
147         trn = Transaction(4, 13, 15, self.d_pub,
    self.m_pub)
148
149         self.assertEqual(self.valid.ledger, [trn])
150
```

```python
# coding=utf-8
import os
import unittest

from src.transactions.transaction import *


class TestTransaction(unittest.TestCase):
    def setUp(self) -> None:
        # load keys
        os.chdir("/home/tcassar/projects/settle/src")
        ldr = keys.RSAKeyLoader()
        ldr.load("./crypto/sample_keys/d_private-key.pem")
        ldr.parse()

        self.key = keys.RSAPrivateKey(ldr)
        self.pub_key = keys.RSAPublicKey(ldr)

        self.trn = Transaction(0, 0, 0, self.pub_key, self.pub_key)
        self.trn.time = 0  # type: ignore

    def test_hash(self):
        """Tests that calling .hash() on a Transaction displays expected behaviour"""
        self.assertEqual(
            self.trn.hash(),
            b"\xb9y\x8f\xe6\xdeo\x08\xd4\x1f\xdd\x01\xa3\xb4\xc5\x1d\x98\xd4\xac\x1bJ\x02\xd3\xa0\x01!\xf3\x99\xdf\xd5\\\x85v",
        )
        self.trn.time = 1
        self.assertNotEqual(
            self.trn.hash(),
            b"\xb9y\x8f\xe6\xdeo\x08\xd4\x1f\xdd\x01\xa3\xb4\xc5\x1d\x98\xd4\xac\x1bJ\x02\xd3\xa0\x01!\xf3\x99\xdf\xd5\\\x85v",
        )

```

```
34      def test_sign(self):
35          """
36          Checks that signing a Transaction displays
    expected behaviour
37          Working on assumption that rsa.Notary is
    working; tested in settle/tests/test_crypto"""
38
39          self.trn.sign(self.key, origin="src")
40
41          with (self.subTest("catch invalid origin"),
    self.assertRaises(ValueError)):
42              self.trn.sign(self.key, origin="no")
43
44          with self.subTest("invalid key types"), self
    .assertRaises(TransactionError):
45              self.trn.sign(12, origin="src")
46              self.trn.sign(self.pub_key, origin="dest
    ")
47
48          with self.subTest("sig_overwrite"), self.
    assertRaises(TransactionError):
49              self.trn.sign(self.key, origin="src")
50
51          with self.subTest("Right sig"):
52              self.assertEqual(
53                  b"\xc1~\xcbu\xecX\x01E*\xf2;O\xe3\
    xf3\x08x\xee\x84\xfc\xe1\xca\x8b\xa2\xedj\xec\x9b\
    xfd\xe5$7\x88n\xb7\x86\xfc~\x98\x91\x80Z\xcd\x1e\xf5
    }\xc2<JS\xefY\xe5UW\xfc\x0e\xd2\xbe\xc9\xea\xfbzm\
    xf2\xa7\x08\x8d\x05\x16\xe4@\xf40\x03I\xca\xbc.\x95\
    xbb\xd3\n\xfd9\xb0Wk\xf4\x03\x96\xdbF\xcd\xa0E\xd1\
    xac\xa6(\xb9\x92\xdb\x841\xe0U\"\xe4\x7f\xeb U\xc9Z\
    xe5\xf6\x19\xc1Wn\x17&\x17\x84Dt\xb6z\xc0\x02\x85`\
    xf3\xd3\xd5\x98t7\xcd_\xfc\xa7\\\xa7t\xe8\xb1\xafl\
    x05\xb3\x07a\xd0jr\xc4}\xd8\xb58\xf4o\x9f\x02\xa2\
    xe6?3B\xf6\xe1\xe6\xb2\x02d\xfd\xd5\x83\xcf\xcc\xb6\
    x06m\xc2p\xd7\x00@\x93H\xc6]\x0c\x98\x1e\xdf\xda\x00
    \x86\xd7\xec\xc7\x10\x025eiry\xd6\x80\xfe\xe2+\xb8\
    x1dn\noB\xc0\xa8a\xc8t\xbfg\xbb\xca(Aj\xa5\xeb\x94\
    x0c-\xacr'\xfe\n^Z\x13\xa0'\xaa\x96{\xf8x\xce\xd6\
    x90",
```

```
54                 self.trn.signatures[self.trn.src],
55             )
56
57     def test_verify(self):
58         """Checks that we are able to verify
   transactions"""
59         # check that we are complained at if no
   valid keys are passed in
60         with self.subTest("catch invalid keys"),
   self.assertRaises(VerificationError):
61             self.trn.verify()
62             self.trn.verify()  # wrong type  # type
   : ignore
63
64         # check works with priv and public keys
65
66         with self.subTest("good verif"):
67             self.trn.sign(self.key, origin="src")
68             self.trn.verify()
69
70         with self.subTest("priv/pub keys"):
71             self.trn.verify()
72             self.trn.verify()
73
74         with self.subTest("verify src, dest"):
75             self.trn.verify()
76             self.trn.verify()
77
78         with self.subTest("verify >1 param"):
79             self.trn.verify()
80
81         with self.subTest("bad key"), self.
   assertRaises(VerificationError):
82             # edit pub key, thus should fail
83             self.pub_key.lookup["n"] = 3
84             self.trn.verify()
85
```