Bioinformatics III

Sixth Assignment

Alexander Flohr (2549738)
Andrea Kupitz (2550260)

June 2, 2018

Exercise 5.1: Boolean Networks

(a) Listing 1 shows the source code of our propagation matrix class, which behaves like such a matrix. Internally it uses a adjacency matrix to efficiently calculate next states. Therefore it depends on the class AdjacencyMatrix shown in Listing 2. Further, the networks states are encoded in the class State, see 3.

Listing 1: Listing of source code

```python
from os.path import exists
from AdjacencyMatrix import AdjacencyMatrix
from State import State

class PropagationMatrix:
    """
    Implements a propagation matrix. The propagated states are not
    preprocessed and are computed on demand to reduce the required
    amount of memory. Thereby increased runtime-changes are negligible
    """
    def __init__(self, filename):
        """
        Create a class object behaving lika a propagation matrix
        for a gene network obtained by textfile
        Textfile specification:
        A > B indicates that gene A turns gene B activate
        A | B indicates that gene A turns gene B inactive
        (space separated)
        """
        # temporary storage for gene lables (to avoid redundancy)
        nodes = set()
        # temporary storage for linkages (to avoid double file reading)
        links = list()
        if exists(filename):
            with open(filename) as openfile:
                for line in openfile:
                    content = line[0:(len(line)-1)].split(" ")
                    nodes.add(content[0])
                    nodes.add(content[2])
                    if (content[1] == ">"):
                        links.append((content[0], content[2], 1))
                    elif (content[1] == "|"):
                        links.append((content[0], content[2], -3))

            # sort genes in alphabetiv order by
            # by making use of list functionalities
            nodes = list(nodes)
            nodes.sort()

            # initialize a adjacency matrix representing the obtained
            # gene network
            self.matrix = AdjacencyMatrix(0, nodes)
            for triple in links:
                self.matrix.setByLable(triple[0], triple[1], triple[2])
        else:
            print(filename, "does not exist")

    def propagate(self, state):
        """
        Derive the propagated state form a given state
        Each gene is set inactive if the scalar product
        of the given state and its column in the adjacency matrix
        is smaller or equal to zero. Otherwise, the gene is set active
        """
        prop_state = State(state.getLables())
        for lab_y in state.getLables():
            # scalar product
            temp = 0
            for lab_x in state.getLables():
                temp += state.getByLable(lab_x) * self.matrix.getByLable(lab_x, lab_y)
            # threshold task specific
            prop_state.setByLable(lab_y, 0 if temp <= 0 else 1)
        return prop_state

    def size(self):
        """
        Returns the number of states
        """
        return self.matrix.size()

    def basinsAndAttractor(self, state):
        """
        Track the states obtained by simulating the regulatory
        network until the states repreat. Return the results divided in
        orbit states and pure basins
        """
        order = list()
        order.append(state.getInt())
        temp = 0
        # simulate network until repetition starts
        while True:
            state = self.propagate(state)
            temp = state.getInt()
            if not (temp in order):
                order.append(temp)
            else:
                break
        # divide states in upper described distinct subsets
        result = [[],[]]
        switch = 0
        for i in range(0, len(order)):
            if order[i] == temp:
                result[0] = order[0:i]
                result[1] = order[i:len(order)]
        return result

    def simplify(self, basatt):
        """
        Simplify redundand and incomplete basins of attractions and
        orbits
        """
        # Since every state can have only one next propagated state,
        # a minimum state-value is unique for every orbit and can be
        # used as key for our simplification
        attracktors = dict()
        basins = dict()
        for ba in basatt:
            temp = min(ba[1])
```

```
                if not (temp in attracktors.keys()):
                    attracktors[temp] = ba[1]
                    basins[temp] = ba[0]
                else:
                    basins[temp].extend(ba[0])
                    basins[temp].extend(ba[1])
            simple = list()
            for i in attracktors.keys():
                simple.append([set(basins[i]), attracktors[i]])
            return simple

        def orbit(self):
            """
            Find basins of attracktion and according orbits
            """
            # Storage for for attracktors and basins
            periodes = list()
            # simulate the network starting from every possile state
            candidates = list(range(0,2**self.matrix.size()))
            while len(candidates) > 0:
                state = State(self.matrix.getLables())
                state.setInt(candidates[0])
                # simulate regulatory network
                temp = self.basinsAndAttractor(state)
                # store obtained results in the according categorie,
                # further remove observed states from the candidate list
                # to minimize runtime
                for basin in temp[0]:
                    if basin in candidates:
                        candidates.remove(basin)
                for attractor in temp[1]:
                    if attractor in candidates:
                        candidates.remove(attractor)
                periodes.append(temp)
            # results might contain reduncdancies, therefore a last
            # simlification step is possible
            return self.simplify(periodes)
```

Listing 2: Listing of source code

```
class AdjacencyMatrix:
    """
    Adjacency matrix, encoding a squared matrix with edge weight
    information between nodes.
    Initilialization and data access is only possible over the
    row and column lables, not by positions
    """

    def __init__(self, initial, lables):
        """
        Initialization of the adjacency matrix, required are an
        initial default weight and row lables, which are also used as
        column lables -> leading to a squared matrix
        """
        self.lables = lables
        # default setup
        self.matrix = [0] * len(lables)
        for i in range(0, len(self.matrix)):
            self.matrix[i] = [initial] * len(lables)

        # Similar to the state class, data access is only possible
        # with row and column lables, dicts enable an acces in linear
        # time
        self.access = dict()
        counter = 0
        for i in self.lables:
            self.access[i] = counter
            counter += 1
    def setByLable(self, a, b, value):
        """
        Set the weight for the edge from a to b
        """
        self.matrix[self.access[a]][self.access[b]] = value

    def getByLable(self, a, b):
        """
        Get the weight for the edge from a to b
        """
        return self.matrix[self.access[a]][self.access[b]]

    def size(self):
        """
        Returns the size of the matrix
        expressed by the number of lables
        """
        return len(self.lables)

    def getLables(self):
        return self.lables

    def show(self):
        """
        Output the matrix as collection of lists
        """
        print("Square Matrix:")
        for i in range(0, len(self.lables)):
            print(self.matrix[i])
```

Listing 3: Listing of source code

```
class State:
    """
    Encode if genes are active (1) or inactive (0) or inactive
    Covers a simple list and extends it with functionality required
    by the PropagationMatrix and SquareMatrix class
    Data acces can only be done by using the gene lables.
    """

    def __init__(self, lables):
        """
        Initialize a state with the set of all gene lables
        """
        self.lables = lables
        # dict allows to access the data by lable in linear time
        self.access = dict()
        counter = 0
        for l in lables:
            self.access[l] = counter
            counter += 1
        # encondes if genes are active or inactive
        self.state = [0] * len(lables)

    def setByLable(self, lable, value):
        """
        Set the gene with a given lable avtive or inactive
        Every input will be translated to active (1) or inactive (0)
        """
        self.state[self.access[lable]] = 0 if value <= 0 else 1

    def getByLable(self, lable):
        """
        Returns 1 if the gene with a certain lable is active,
        0 otherwise
        """
        return self.state[self.access[lable]]
```

```python

    def getLables(self):
        """
        Returns the lables of genes
        """
        return self.lables

    def size(self):
        """
        Returns the lenght of the state,
        i.g. the number of encoded genes
        """
        return len(self.lables)

    def getInt(self):
        """
        Returns the unique integer obtained by the binary encoded
        genes (active or inactive)
        """
        value = 0
        for n in range(0,len(self.lables)):
            value += self.state[n] * (2**n)
        return value

    def setInt(self, value):
        """
        Initializes a state whose binary representation equals the
        provided integer value
        """
        if value == 0:
            return
        binaries = 2**(len(self.lables)-1)
        pos = len(self.lables) - 1
        while pos >= 0:
            if value >= binaries:
                self.state[pos] = 1
                value -= binaries
            binaries /= 2
            pos -= 1
        return

    def show(self):
        """
        Output the state as binary list
        """
        print("State:")
        print(self.state)
```

(b) Listing 4 shows source code applying the the functiionality of the code shown in Listing 1, which includes the network simulation.

1) It makes sense to stop the propagation when a state is observed a second time, from then on we will only observe orbiting behavior of the network states. The results of 2) are shown in this way. e.g. the first repeting state is the last shown.

2) Programs output for the required initial states:

```
Initial state 1:
1 -> 3 -> 7 -> 23 -> 55 -> 63 -> 13 -> 1

Initial state 4:
4 -> 18 -> 36 -> 26 -> 4

Initial state 21:
21 -> 51 -> 47 -> 13 -> 1 -> 3 -> 7 -> 23 -> 55 -> 63 -> 13

Initial state 33:
33 -> 11 -> 5 -> 19 -> 39 -> 31 -> 5
```

Listing 4: Listing of source code

```python
from PropagationMatrix import PropagationMatrix
from State import State

def trackPropagation(state, repeats):
    """
    Visualize the propagations by a sequence of integers
    """
    track = str(state.getInt())
    for i in range(1, repeats):
        state = prop.propagate(state)
        track += " -> "
        track += str(state.getInt())
    print(track)

# Initialize propagation network  with text file
# File contains structural informations of the given
# gene regulatory network
prop = PropagationMatrix("net.txt")

# initialize with the state integer 13
state_a = State(['A','B','C','D','E','F'])
print("                        Exercise 6.1 b)                        ")
print("---------------------------------------------------------------")
print("\nInitial state 1:")
state_a.setInt(1)
trackPropagation(state_a, 8)

# initialize with the state integer 13
print("\nInitial state 4:")
state_b = State(['A','B','C','D','E','F'])
state_b.setInt(4)
trackPropagation(state_b, 5)

# initialize with the state integer 13
print("\nInitial state 21:")
state_c = State(['A','B','C','D','E','F'])
state_c.setInt(21)
trackPropagation(state_c, 11)

# initialize with the state integer 13
```

```
print("\nInitial state 33:")
state_d = State(['A','B','C','D','E','F'])
state_d.setInt(33)
trackPropagation(state_d, 7)

print()
print("                        Exercise 6.1 c)                      ")
print(" -------------------------------------------------------------- ")
orbits = prop.orbit()
for i in range(0, len(orbits)):
    print()
    length = len(orbits[i][1])
    print("Orbit " + str(i + 1) + " with length " + str(length) + ":")
    print(orbits[i][1])
    print("Set of basins:")
    print(orbits[i][0])
    coverage = float(len(orbits[i][0]))
    coverage /= float(2**prop.size())
    coverage *= 100.0
    print("Relative coverage: " + str(coverage) + "%")
```

(c) Output of the progam listing the orbits:

```
Orbit 1 with length 1:
0
Set of basins:
0, 6, 8, 12, 16, 20, 22, 24, 28, 32, 34, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58,
60, 62
Relative coverage:  35.9375%

Orbit 2 with length 7:
1, 3, 7, 23, 55, 63, 13
Set of basins:
1, 3, 7, 9, 13, 21, 23, 25, 29, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63
Relative coverage:  32.8125%

Orbit 3 with length 4:
4, 18, 36, 26
Set of basins:
2, 4, 36, 38, 10, 14, 18, 26, 30
Relative coverage:  14.0625%

Orbit 4 with length 4:
5, 19, 39, 31
Set of basins:
33, 35, 5, 37, 39, 11, 15, 17, 19, 27, 31
Relative coverage:  17.1875%
```

(d) Interpretation:
1) ???
2) Two special are gene A and D. If A is active, the network becomes permanent ïmpulses; keeping the network in motion. Further, D hinders the network to remain fully active, i.g. if gene D is activated, it throws the network back into an earlier state, initiaing the orbiting behavior. This is further ensured since D inactivates its own activating gene.
Table 1 shows the two shorter orbits including the genes activation status at each state of the network. The upper described principles, are demonstrated for Orbit 4, where Ä activates the network step by stepälong the regulatory linkages until D gets activated. Afterwards, D inactivates most genes to reset the network to an earlier state.
For Orbit 3, the driving mechanisms are different. Here, B and C activate each other so

Table 1: Comparison of the two orbits with length 4

Orbit 3							Orbit 4						
Decimal	A	B	C	D	E	F	Decimal	A	B	C	D	E	F
4			X				5	X		X			
18		X			X		19	X	X			X	
36			X			X	39	X	X	X			X
26		X		X	X		31	X	X	X	X	X	

that only one of both is active at the same time. When C is active, it further activates E what initiates D to inhibit B, E and F after 3 propagations. Since this inhibition occurs when C is active, the orbit closes and starts again. If D would have even distance (number of activating forward linkages), D would inhibit the active B what would turn the complete network inactive. In this case we would not observe orbiting behavior.

Exercise 5.2: Differential Expression Analysis