Rahul Vaidun
rvaidun@ucsc.edu
1 April 2021

CSE 13S Spring 2021
Assignment 4: The Circumnavigations of Denver Long
Design Document

I. Description

This program uses depth-first search to find the shortest hamiltonian path with a given graph.

II. Pseudocode

A. BitVector

```
class BitVector:
   def __init__(self, length):
       self.length = length
       self.vector = [0] * length

   def bv_length(self):
       return self.length

   def bv_set_bit(self, i):
       bytepos = i // 8
       bitpos = i % 8
       self.vector[bytepos] = ((1 << bitpos) | self.vector[bytepos])

   def bv_clr_bit(self, i):
       bytepos = i // 8
       bitpos = i % 8
       self.vector[bytepos] = (self.vector[bytepos] & ~(1 << (bitpos)))

   def bv_get_bit(self, i):
       bytepos = i // 8
       bitpos = i % 8
```

```
        return (self.vector[bytepos] >> bitpos) & 1

    def bv_xor_bit(self, i, bit):
        bytepos = i // 8
        bitpos = i % 8
        b = self.bv_get_bit(i)
        # clear bit and then set it to either 0 or 1
        self.vector[bytepos] = (self.vector[bytepos] & (
            ~(1 << bitpos))) | ((b ^ bit) << bitpos)

    def bv_print(self):
        print(self.vector)
```

B. BitMatrix

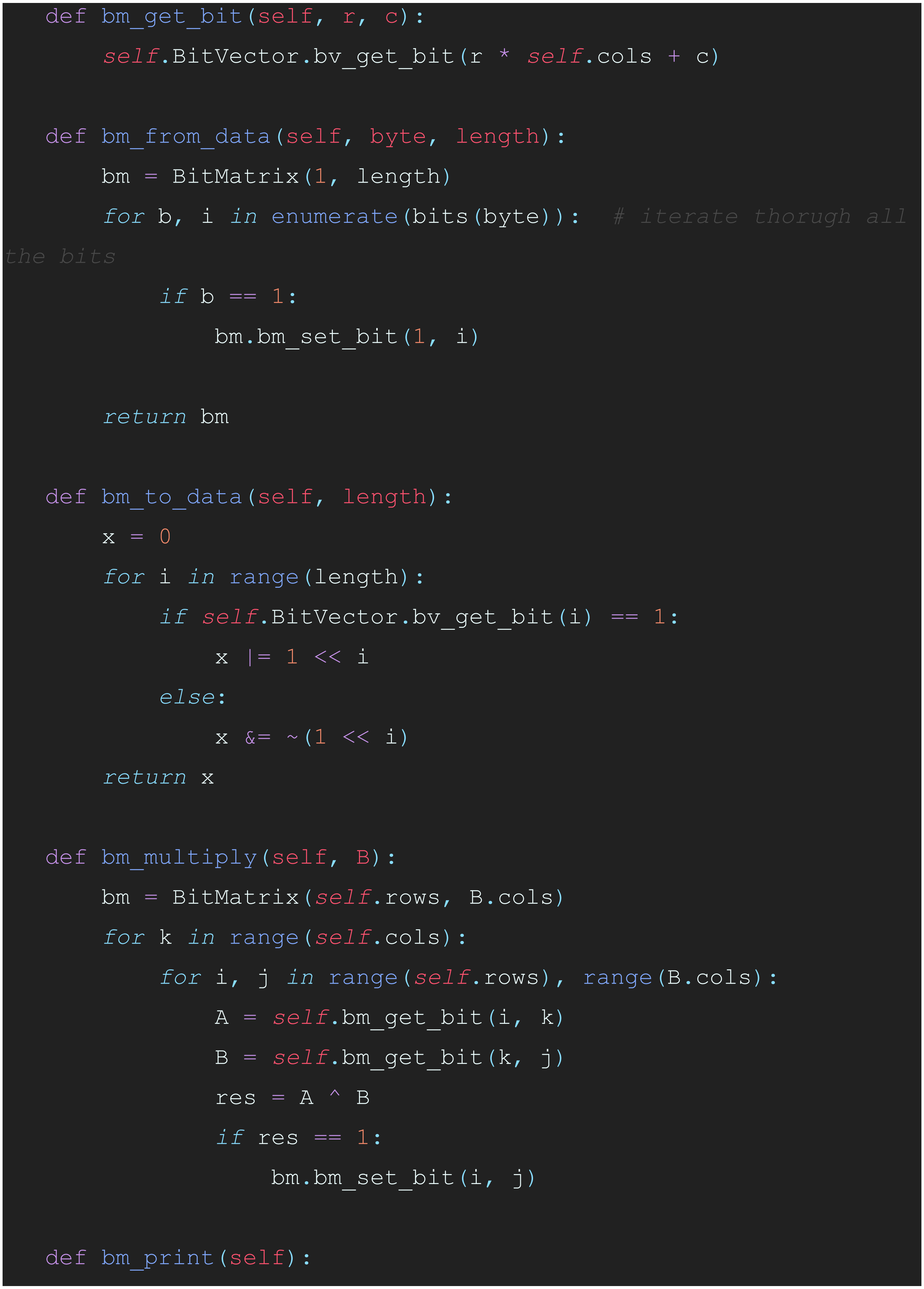
```
class BitMatrix():
    def __init__(self, rows, cols):
        self.rows = rows
        self.cols = cols
        self.BitVector = BitVector(rows * cols)

    def bm_rows(self):
        return self.rows

    def bm_cols(self):
        return self.cols

    def bm_set_bit(self, r, c):
        self.BitVector.bv_set_bit(r * self.cols + c)

    def bm_clr_bit(self, r, c):
        self.BitVector.bv_clr_bit(r * self.cols + c)
```



```
    def bm_get_bit(self, r, c):
        self.BitVector.bv_get_bit(r * self.cols + c)

    def bm_from_data(self, byte, length):
        bm = BitMatrix(1, length)
        for b, i in enumerate(bits(byte)):  # iterate thorugh all the bits
            if b == 1:
                bm.bm_set_bit(1, i)

        return bm

    def bm_to_data(self, length):
        x = 0
        for i in range(length):
            if self.BitVector.bv_get_bit(i) == 1:
                x |= 1 << i
            else:
                x &= ~(1 << i)
        return x

    def bm_multiply(self, B):
        bm = BitMatrix(self.rows, B.cols)
        for k in range(self.cols):
            for i, j in range(self.rows), range(B.cols):
                A = self.bm_get_bit(i, k)
                B = self.bm_get_bit(k, j)
                res = A ^ B
                if res == 1:
                    bm.bm_set_bit(i, j)

    def bm_print(self):
        print(self.BitVector)
```

C.

III.