CATAM Part II - 17.3 - Hamiltonian Cycles

Introduction

Our first task is to create a sensible data structure to store our graph in, and also write a function that generates a random graph from $\mathcal{G}(n,p)$. Since I'm coding in python, I've implemented a Graph class, which stores all our graph's information, and will contain various methods pertaining to our graph. It allows for us to generate a random graph or input a vertex and edge set.

Question 1

The simplest, and least efficient way to check if a graph has a Hamiltonian cycle is to check each of the $n!$ individual permutations of the vertex set. We can reduce checking slightly since cyclic permutations are equivalent, so we only have $(n-1)!$. The hard part here is iterating over all permutations, since there's eventually too many to store in memory. We can overcome this by, say, Heap's Algorithm[1], but our implementation uses the *permutations* iterator from the standard python library *itertools*. It can be found under *naive_hamiltonian*1() as a method of the Graph class. We find this does work, but is exceedingly slow for $n > 12$ so unable to generate the results we want.

A slightly smarter simple approach is as follows. We try to generate paths starting at 1, using the minimum possible vertex at each step. If we hit a dead end, we go back and try the next minimum vertex. This eventually gives every path starting at 1, so must give us a cycle if it exists. Otherwise it will run through all possible paths and return to the 1 vertex path, in which case there is no cycle. The algorithm is as follows

First set **candidate** = [1], **max-candidate** = [1,0,0, . . . , 0] and **k** = 1. **k** is our track of which position vertex in the path we're currently looking for (using 0-indexing) and **max-candidate** records where we've been: The algorithm is as follows:

Loop the following

1) If |**candidate**| = n, check if the final vertex is a neighbour of 1, and if so return **candidate** since the cycle is valid. If it's not a neighbour of 1, define **maxcandidate** = **candidate**, remove the last element of **candidate**, and reduce **k** by 1.

2) Now |**candidate**| < n. We find the minimal vertex we may append to candidate that we havn't already tried (by picking the minimal neighbour of the most recent vertex greater than the **k**'th element of **maxcandidate**), and append it to **candidate**. Set **maxcandidate** to **candidate** + [0] to indicate we havn't tried any position k+1 elements yet. Iterate **k** by 1.

3) If no such possible neighbour exists, set **maxcandidate** to **candidate**, remove the most last element of **candidate**, and reduce **k** by 1.

4) We terminate if we ever get a successful cycle, or if **candidate** ever returns to [1] (or possibly [], if 1 had no neighbours) after this process, ie if after checking all paths we fail to find a cycle.

Our implementation is *simple_hamiltonian*(). This is just about quick enough, and we get the following data, gathered by *q*1().

```
* Order=3, Size=3, {1: {2, 3}, 2: {1, 3}, 3: {1, 2}} * has a hamiltonian cycle:
   [1, 2, 3]
* Order=3, Size=2, {1: {2}, 2: {1, 3}, 3: {2}} * has no hamiltonian cycle
* Order=4, Size=3, {1: {2, 3}, 2: {1, 3}, 3: {1, 2}, 4: set()} * has no
   hamiltonian cycle
* Order=6, Size=8, {1: {2, 4}, 2: {1, 4, 5}, 3: {4, 6}, 4: {1, 2, 3}, 5: {2, 6},
    6: {3, 5}} * has a hamiltonian cycle: [1, 2, 5, 6, 3, 4]
```

[1] https://en.wikipedia.org/wiki/Heap%27s_algorithm

p/n	**4**	**6**	**8**	**10**	**12**	**14**	**16**	**18**	**20**
0.1	6	0	0	0	0	0	0	0	0
0.3	2	3	5	10	19	35	53	71	85
0.5	14	26	51	76	92	97	99	100	100
0.7	44	75	95	99	99	99	99	100	100
0.9	84	98	99	100	100	100	100	100	100

Table 1: Number of graphs containing Hamiltonian cycles from a selection of 100 taken from $\mathcal{G}(n,p)$

a/n	**4**	**6**	**8**	**10**	**12**	**14**	**16**	**18**	**20**
0.1	0	0	0	0	0	0	0	0	0
0.55	0	0	0	0	0	0	0	0	0
1	5	2	1	2	0	0	1	1	1
1.45	18	13	14	18	24	20	22	29	32
1.9	36	43	51	66	64	66	62	69	100

Table 2: Number of graphs containing Hamiltonian cycles from a selection of 100 taken from $\mathcal{G}(n, a\log n/n)$

Question 2

Suppose we have n vertices. The worst case is we have no Hamiltonian cycle. Assume each of the $(n-1)!$ paths are checked, only failing at the last step (ie every length n path is Hamiltonian starting at v_0, but not a cycle) We assume each simple operation, eg iterating, looking up a value in a list take time c. For each. At worst, we need about $2n+10$ simple operations per iteration, of which most are spent calculating the new possible vertices to append. This gives a time of $c(2n+10)$ per permutation. So we end up with a total running time of $(n-1)!(2n+10)c = O(n!)$.

For an average case, it'll be useful to know how many Hamiltonian cycles we should expect to have in a $\mathcal{G}(n,p)$ graph. This isn't too hard to show:

Theorem. *The expected number of hamiltonian cycles in a random graph $\mathcal{G}(n,1/2)$ is given by $\frac{1}{2}(n-1)!p^n$, for $n \geq 3$.*

Proof. First, K_n has $\frac{1}{2}(n-1)!$ cycles of length n, all of which are Hamiltonian (start at 1, have n-1 choices for the second vertex, n-2 for the third... Factor of 1/2 comes from the undirectedness of the graph). Order these cycles, and let X_i be the event G contains the i'th cycle. Then $\mathbb{E}(X)$ is the expected number of hamiltonian cycles in G, and by linearity of expectation is equal to $\sum_i \mathbb{E}(X_i)$. Now $\mathbb{E}(X_i) = p^n$ since each edge occurs independently with probability p, giving our result. □

Suppose an average graph can be taken from $\mathcal{G}(n,1/2)$. There are $(n-1)!$ paths, of which $(n-1)!/2^{n+1}$ are Hamiltonian, so we expect our 2^{n+1}'th length n path to be Hamiltonian. To get from one length n path to the next is constant time in this regime, and about 6 simple operations. To build our first cycle, we expect it to take at most 2n steps, each of which is about 6 simple operations. So we get a running time of about $(12n + 6 \times 2^{n+1})c = O(2^{n+1})$, at least for n large. This is exponential in n.

Question 3

Modifying the code slightly, to print the number of graphs G with $\delta(G) < 2$, we find most, at least for $a < 1.45$ for a defined in Table 2, graphs fail to be Hamiltonian because $\delta(G) < 2$. The second range may well have been chosen because of the following theorem, proved in IID Graph Theory.

Theorem. *Let $\omega(n) \to \infty$. If $p = \frac{\log(n)-\omega(n)}{n}$ then G has isolated vertices almost surely. If $p = \frac{\log(n)+\omega(n)}{n}$ then G has no isolated vertices almost surely.*

Proof. Note if $X = \Sigma_A I_A$ is a sum of indicator functions then $\mathrm{Var}(X) = \Sigma_{A,B}\mathbb{P}(A)[\mathbb{P}(B \mid A) - \mathbb{P}(B)]$ simply by expanding.

Now let X be the number of isolated vertices $X = \Sigma_v I_v$ where I_v indicated v being isolated. So

$$\begin{aligned}\mathrm{Var}(X) &= \Sigma_{u,v}\mathbb{P}(u \text{ isolated})[\mathbb{P}(vu \text{ isolated} \mid uu \text{ isolated}) - \mathbb{P}(v \text{ isolated})] \\ &= (1-p)^{n-1}[1-(1-p)^{n-1}] + n(n-1)(1-p)^{n-1}[(1-p)^{n-2} - (1-p)^{n-1}] \\ &\leq \mathbb{E}(X) + n^2(1-p)^{n-1}(1-p)^{n-2} \\ &= \mathbb{E}(X) + \frac{p}{1-p}(\mathbb{E}(X))^2\end{aligned}$$

where the first term comes from u and v being the same, and the second term otherwise. Now if $p = \frac{\log(n)+\omega(n)}{n}$

$$\mathbb{E}(X) = \frac{1}{1-p}n(1-p)^n \leq \frac{1}{1-p}ne^{-pn} \to 0$$

So $X = 0$ a.s. by Markov's Inequality. If $p = \frac{\log(n)-\omega(n)}{n}$

$$\mathbb{E}(X) \approx \frac{1}{1-p}ne^{-pn} \to \infty$$

So

$$\frac{\mathrm{Var}(X)}{(\mathbb{E}(X))^2} \leq \frac{1}{\mathbb{E}(X)} + \frac{p}{1-p} \to 0$$

So $X \neq 0$ a.s. by Chebyshev's Inequality.

$\square$

Here we're considering a range of values $p \in [\frac{\log(n)-0.9\log(n)}{n}, \frac{\log(n)+0.9\log(n)}{n}]$, so as n grows, the probability of having isolated vertices for $a < 1$ grows tends to 1, which agrees with our results.

Question 4

The *smarter_hamiltonian*(T) method of the Graph class performs the algorithm. To check it works here's what we get on the previous small examples:

```
* Order=3, Size=3, {1: {2, 3}, 2: {1, 3}, 3: {1, 2}} * has a hamiltonian cycle:
   [1, 3, 2]
* Order=3, Size=2, {1: {2}, 2: {1, 3}, 3: {2}} * has no hamiltonian cycle
* Order=4, Size=3, {1: {2, 3}, 2: {1, 3}, 3: {1, 2}, 4: set()} * has no
   hamiltonian cycle
* Order=6, Size=8, {1: {2, 4}, 2: {1, 4, 5}, 3: {4, 6}, 4: {1, 2, 3}, 5: {2, 6},
    6: {3, 5}} * has a hamiltonian cycle: [1, 2, 5, 6, 3, 4]
```

To determine what a suitable T would be for various n,p we'll run some trials, and determine a value of T such that we'll find 95% of hamiltonian cycles. We can do so by first using our *simple_hamiltonian*() function to only pick out graphs with hamiltonian cycles, then use, say *smarter_hamiltonian*(1000), assuming all would be found within 1000 iterations, to determine a 95'th percentile on T. We'll run 5000 trials, by which I mean we'll test 5000 graphs with hamiltonian cycles for each n,p. We'll only go up to $n = 14$, since *simple_hamiltonian*() is fairly slow for larger n for small p, and won't bother testing $p < 0.3$ since we rarely ever have hamiltonian. We gather the following data using $q4()$.

p/n	**4**	**6**	**8**	**10**	**12**	**14**
0.3	6	15	33	62	80	103
0.5	7	16	27	35	34	33
0.7	7	13	16	17	18	19
0.9	6	8	10	18	14	15

Table 3: 95'th percentile of T required to find a hamiltonian cycle, given one exists, from a selection of 5000 trials in $\mathcal{G}(n,p)$

We find unsurprisingly a larger T is required for small p, and not too large a T is required, making this fairly fast. If we only have T depending on n, setting T to about n^2 should catch almost all Hamiltonian cycles, and is a lot better than the average case exponential complexity.

Question 5

Without any advanced modelling it seems n/p^2 is fairly decent, and certainly gives a T larger than all of the above 0.95 percentile value.

In general we expect small graphs with small p to have few Hamiltonian cycles and large p to have many. The algorithm terminates in 2 cases:

1) If v_0 has no neighbours
2) If we find a hamiltonian path
3) If we hit T iterations

Case 1 is unlikely in general, though occurs more for p small than p large. Case 2 occurs more often for p large, and case 3 for p small, so we expect the algorithm to take much longer for small p, as case 3 takes much longer than case 3.

In the average case, we expect the 2^{n+1}th length n path to be a cycle. Once we reach length n, step 3 is constant time, since all we're doing is a look up and a splice. Before we reach length n, we use step 2, which is also constant time. We don't expect to take more than 2 steps to find a neighbour of v_k not in P_j (since we have p=1/2), and step 2 itself is $O(n)$ (since we're reading the set of neighbours, and intersecting it with P_j). If step 2 fails to find a valid neighbour, we go on to the constant time step 3. So this step is in total $O(n)$. So in average case, the algorithm is $O(2^{n+1})$ to find a hamiltonian cycle, just as the previous algorithm.

Code

```
import itertools
import math
import random
from collections import defaultdict

import numpy


def nCr(n, r):
    f = math.factorial
    return int(f(n) / f(r) / f(n - r))


# import numpy

class Graph(object):
    """ Graph data structure """

    def __init__(self, vertices=[], edges=[], random=False, n=0, p=0):
        """vertices: list of vertices of graph
           edges: list of tuples giving edges of graph
           random: indicates we want to generate a random graph
           n: [n] forms vertex set of random graph
           p: probability any given edge in our random graph
           """

        self._graph = defaultdict(set)

        if random:
            vertices = list(range(1, n + 1))
            random_indices = list(numpy.random.binomial(1, p, size=nCr(n, 2)))
            all_edges = []
            for i in range(len(vertices)):
                for j in range(i + 1, len(vertices)):
                    all_edges.append((vertices[i], vertices[j]))
            edges = [all_edges[i] for i in range(len(all_edges)) if random_indices[i] == 1]

        self.add_edges(edges)
        self.order = len(vertices)
        self.size = len(edges)
        self.vertices = vertices
        self.edges = edges

        # calculate  delta
        delta = self.order - 1
        for v in self.vertices:
            deg_v = len(self._graph[v])
            if deg_v < delta:
                delta = deg_v
        self.delta = delta

    def __str__(self):
        """Print the graph in a sensible form"""
        return "* Order=" + str(self.order) + ", Size=" + str(self.size) + ", " + str(
            dict(self._graph)) + " *"

    def add_edges(self, edges):
        """ Add edges (list of tuple pairs) to graph """
        for v1, v2 in edges:
            self.add(v1, v2)

    def add(self, v1, v2):
        """ Add connection between node1 and node2 """
        self._graph[v1].add(v2)
        self._graph[v2].add(v1)

    def neighbours(self, v):
        return self._graph[v]

    def naive_hamiltonian(self):
        permutations = itertools.permutations(self.vertices[1:])
        for permutation in permutations:
            if permutation[-1] not in self.neighbours(self.vertices[0]):
                continue
            for i in range(len(permutation) - 1):
                if permutation[i + 1] not in self.neighbours(permutation[i]):
                    break
            return [self.vertices[0]] + list(permutation)
        return

    def simple_hamiltonian(self):
        candidate = [self.vertices[0]]
        maxcandidate = [1] + [0] * (self.order - 1)
        position = 1
        while True:

            if len(candidate) == self.order:
                if candidate[-1] in self.neighbours(self.vertices[0]):
                    return candidate
                else:
                    maxcandidate = candidate
                    candidate = candidate[:-1]
                    position -= 1
                    # print(position)
                    continue
            else:

                neighbours = self.neighbours(candidate[position - 1])
                compareto = maxcandidate[position]
                possible_new_vertices = [y for y in [x for x in neighbours if x not in candidate] if
                                         y > compareto]

                if bool(possible_new_vertices):
                    newvertex = min([y for y in neighbours if y > compareto and y not in candidate])
                    candidate.append(newvertex)
                    maxcandidate = candidate + [0]
                    position += 1
                else:
                    maxcandidate = candidate
                    candidate = candidate[:-1]
                    # print(candidate)
                    position -= 1

                    if candidate == [self.vertices[0]] or candidate == []:
                        return False

    def smarter_hamiltonian(self, T):
        P = [self.vertices[0]]
        count = 1

        while True:
            if len(P) == self.order and P[-1] in self.neighbours(self.vertices[0]):
                return P, count

            elif len(P) < self.order and bool(
                    self.neighbours(P[-1]).intersection(
                        set(self.vertices) - set(P))):  # set non empty
                P.append(random.sample(
                    (self.neighbours(P[-1]).intersection(set(self.vertices) - set(P))), 1)[0])
            else:
                vk_neighbours = self.neighbours(P[-1])
                if not bool(vk_neighbours):
                    return False, count  # zero degree of vk
                vi = random.sample(vk_neighbours.intersection(set(P)), 1)[0]
                i = P.index(vi)
                new_P_start = P[:i + 1]
                new_P_end = list(reversed(P[i + 1:]))
                P = new_P_start + new_P_end
            count += 1

            if count > T:
                return False, count


def q1():
    # First some paticular graphs that we either know have hamilton cycles or don't
    for graph in [Graph(vertices=[1, 2, 3], edges=[(1, 2), (2, 3), (3, 1)]),
                  Graph(vertices=[1, 2, 3], edges=[(1, 2), (2, 3)]),
                  Graph(vertices=[1, 2, 3, 4], edges=[(1, 2), (2, 3), (3, 1)]),
                  Graph(vertices=[1, 2, 3, 4, 5, 6],
                        edges=[(1, 2), (3, 4), (2, 4), (1, 4), (5, 6), (6, 3), (2, 5), (5, 6)])]:
        result = graph.simple_hamiltonian()
        if result:
            print(str(graph) + " has a hamiltonian cycle: " + str(result))
        else:
            print(str(graph) + " has no hamiltonian cycle")

    pvalues = [0.3]
    nvalues = list(range(4, 21, 2))

    trials = 100

    factorvalues = [0.1, 0.55, 1, 1.45, 1.9]
    for n in nvalues:
        for factor in factorvalues:
            count = 0
            deltacount = 0
            for trial in range(trials):
                graph = Graph(random=True, n=n, p=factor * numpy.log(n) / n)
                if graph.simple_hamiltonian():
                    count += 1
                if graph.delta < 2:
                    deltacount += 1
            print("n=" + str(n) + ", factor=" + str(factor) + ", hamiltonian: " + str(
                count) + ", non-hamiltonian: " + str(trials - count) + ", delta<2 :" + str(
                deltacount))


def q4():
    for graph in [Graph(vertices=[1, 2, 3], edges=[(1, 2), (2, 3), (3, 1)]),
                  Graph(vertices=[1, 2, 3], edges=[(1, 2), (2, 3)]),
                  Graph(vertices=[1, 2, 3, 4], edges=[(1, 2), (2, 3), (3, 1)]),
                  Graph(vertices=[1, 2, 3, 4, 5, 6],
                        edges=[(1, 2), (3, 4), (2, 4), (1, 4), (5, 6), (6, 3), (2, 5), (5, 6)])]:

        result, T = graph.smarter_hamiltonian(1000)
        if result:
            print(str(graph) + " has a hamiltonian cycle: " + str(result))
        else:
            print(str(graph) + " has no hamiltonian cycle")

    trials = 1000
    pvalues = [0.3, 0.5, 0.7, 0.9]
    nvalues = list(range(4, 15, 2))
```

```
trials = 10000

for n in nvalues:
    for p in pvalues:
        count = 0
        Tvalues = []
        while count < 5000:
            graph = Graph(random=True, n=n, p=p)
            result = graph.simple_hamiltonian()
            if result:
                count += 1
                found, T = graph.smarter_hamiltonian(1000)
                if found:
                    Tvalues.append(T)
        print("n=" + str(n) + ", p=" + str(p) + ", #hamiltonian cycles: " + str(
            count) + ", 95'th T percentile: " + str(numpy.percentile(Tvalues, 95)))
```