Computationally Finding Square-Sum Cycles modulo m

by Naing Lin Tun
May 31st, 2018

Contents

Acknowledgements

I would like to give special thanks to the following:

- Professor Andrew Shallue from Illinois Wesleyan University for mentorship and advice on our research project in his MATH 370 course.

- Patrick Ward, my project partner, for his contribution towards his mathematical findings and great ideas.

- Isaac Dragomir for providing information about his previous talk on the similar topic "The Square-Sum Problem" and more.

1 Introduction

The project idea is heavily influenced by the Numberphile video entitled *The Square-Sum problem*. It says that:

> Given a sequence of consecutive positive integers from 1 to n inclusively, can we order this sequence in such a way that the sum of each consecutive equals to a square? Symbolically, this problem can be presented as:
>
> $$x + y = k^2$$
>
> where $x, y \in \mathbb{Z}^+$ are two consecutive terms in the sequence and $k \in \mathbb{Z}$.

The video gives an example of $n = 15$ and the answer is:

$$8, 1, 15, 10, 6, 3, 13, 12, 4, 5, 11, 14, 2, 7, 9$$

It uses the graph theory approach where there are n vertices and each integer in the sequence $\{1, \ldots, n\}$ is a vertex. Then, we draw an edge between two vertices if and only if the above equation is satisfied. If there exists a Hamiltonian path in our graph, we have found our desirable ordered-sequence.

We transformed the idea of Square-Sum to Square-Sum modulo m and the equation is as follows:

$$x + y \equiv k^2 \pmod{m}$$

where $m \in \mathbb{Z}^+$. In addition, instead of just finding a Hamiltonian path from the graph generated, we intend to find whether there can exist a Hamiltonian cycle for a particular n and m.

Drawing a graph for large value of n (number of vertices) would be very inconvenient to do it by hand, let alone to find a Hamiltonian cycle within it. Thus, we plan to develop a computer program which would help us accomplish what we desire. It is relatively fast and efficient to draw graphs, but it is a different story for finding Hamiltonian cycle.

Since Hamiltonian cycle finder (the decision version) is an NP-Complete problem, there is no known fast polynomial time algorithm for it. Thus, our original intent for the research project was to find ways to optimize the algorithm for our particular kind of graphs and gather data for large graphs. In the process of doing so, we stumbled among some interesting results, such as for $n = m$ where n is primes greater than 9, there always exist a cycle. Even though there is a $O(n^2 2^n)$ dynamic-programming algorithm, we focus to optimize the $O(n \cdot n!)$ brute-force backtracking algorithm because it gives us more vivid patterns of cycle formation in the graphs.

2 Definitions

Definition 1. *Congruence modulo: Let $a, b, m \in \mathbb{Z}$. We say that a and b are congruent modulo m iff $m \mid (a - b)$. Symbolically, we say $a \equiv b \pmod{m}$.*

There are various ways to present the definition of congruence modulo. Another way to define it is that $a \equiv b \pmod{m}$ if and only if $a\%m = b\%m$ where $\%$ is the modulo operator. In other words, the remainder of a/m should equal the remainder of b/m.

Definition 2. *Residue set: The square residue set of mod m, denoted $R(m)$ is the set $\{k^2\%m \neq 0 : k \in \mathbb{Z}^+ \text{ and } 1 \leq k \leq m/2\}$.*

Definition 3. *Adjacency matrix: It is a (boolean) matrix representation of the graph. For a simple undirected graph, if there is an edge between x and y vertices, then the x^{th} row and y^{th} column of the matrix is 1 and also, x^{th} column and y^{th}row is 1; otherwise, 0.*

An example is shown below (source: TutorialRide):

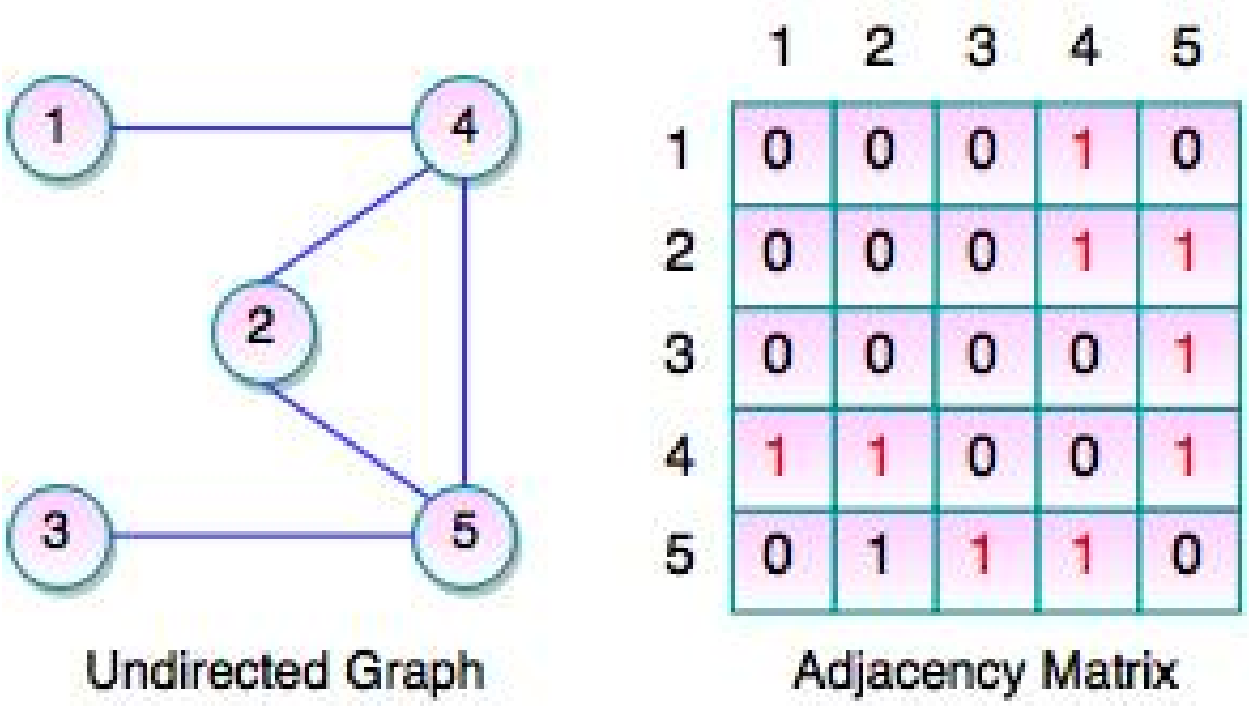


Fig. Adjacency Matrix Representation of Undirected Graph

Definition 4. *Hamiltonian path: It is a path that visits every vertex of the graph exactly once.*

Definition 5. *Hamiltonian cycle: It is a cycle which visits every vertex of the graph exactly once, except the stat note which gets visited twice (at the start and at the end). In this paper, it is suffice to know that Hamiltonian cycle is a Hamiltonian path whose start and end vertices is also connected via an edge.*

Note that for a Hamiltonian cycle to exist, the degree of every vertex should be at least 2. In addition, because it's a cycle, the start point does not matter, unlike Hamiltonian path.

3 Optimizing algorithms

I wrote a computer program in C++ to test and generate multiple data for our research project. Please find my program on GitHub project page (please make sure to refer to README document if you plan to use the same program). I also pull out some parts of the code in the following section to discuss optimization techniques in details and finding patterns.

3.1 Calculating the residue set of mod m

The intuitive idea to calculate square mod m is to take the set $\{1, 2, \ldots, m/2\}$, square each integer in the set and take mod of m. Then, we store it in some data structure, either in a set or vector. As we linearly calculate square mod m, the efficiency will depend on the data structure that we use to store the results. A set would not allow any repetitive elements, which is exactly what we want, but it would also automatically call a sorting algorithm for each new appended data. The complexity of the algorithm will become $O(m^2 \cdot log\ m)$, assuming that the sorting algorithm has $O(m \cdot log\ m)$ worst-case runtime. On the other hand, if we want to use vector, we would need to avoid storing repetitive results. Thus, whenever we want to append a new data into the vector, we would have to linearly scan the vector first to check for repetitions. Thus, the algorithmic complexity using a vector will be $O(m^2)$.

However, some optimizations can be done. The following optimized function, implemented in C++, has the complexity of $O(m)$:

```
vector<int> calcSquareMod(int mod_num) {
  /** This algorithm calculates the residue of square % mod_num excluding 0
   *  For instance: if mod_num = 5:
   *    the residue of squares are mod of 1, 4, 9 16, 25, ....
   *    the residue set would be {1, 4} excluding 0
   *
   *  @param: int mod_num
   *  @return: vector of residue set of squares
```

```
   *
   */

  bool residue[mod_num];
  vector<int> squareMod; // the residue set square mod

  for (int i = 1; i < mod_num; i++) {
    residue[i] = 0;
  }

  int x1=0, x2=0;
  int count = 0;
  for (int i = 1; i <= mod_num/2; i++) {
    int val = (i*i)%mod_num;
    x1 = val;

    if (x1 == x2 || (count >= 2 && val == squareMod[count-2])) {
      // The condition is for optimizing the algorithm.
      // Since the residue set (in modular arithmetic, Z/nZ) has a cyclic
      // pattern, we can break the loop if we detect "repetitions".
      // Repetition can occur in the following two forms:
      // 1. (expressed by the first disjunct of the condition):
      //    when two consecutive residue are equal
      //    for instance: (1^2) mod 3 = (2^2) mod 3.
      //    Then, the residue set is non other than {1} only.
      // 2. (expressed by the second disjunct of the condition):
      //    when the residue value "goes backward"
      //    for instance, square mod 7 = {1, 4, 2, 4, 1, 4, 2, ....}
      //    We can see the cyclic pattern. Thus, the residue set is just
      //    {1, 4, 2}
      break;
    }

    if (residue[val] != 1 && val != 0) {
      // if the residue value is not already in the set (to avoid repetition)
      // and excluding 0 from the residue set
      residue[val] = 1;
      x2 = val;
      squareMod.push_back(val);
      count++;
    }
  }

  return squareMod;
}
```

The key idea for this optimization is to detect the "cyclic" behavior of the residue value as stated by the comments in the code. Instead of iterating through the whole $\{1, \ldots, m/2\}$ set, the algorithm will break as soon as it detects a repetitive behavior, thereby further

making the algorithm more efficient.

3.2 Calculating adjacency matrix of the graph

The first step to compute an adjacency matrix is to create a 2-dimensional array to store the matrix. The brute-force approach would be to try to compute the sum of every pair of vertices % m and if it equals to one of the values from the residue set, then we connect those two vertices via an edge.

For instance, let $m = 5$ and $n = 5$. The residue set is $\{1, 4\}$. We want to check the connectivity between edge 1 and 3. Then, (1+3)%5 = 4, which is one of the values in the residue set. Thus, we set $adjMat[1-1][3-1] = 1$. We do "-1" for indexes because arrays start from index 0.

This brute-force algorithm will have the complexity of $O(n^3)$ – adjacency matrix itself has the size of n^2 and the size of residue set has a bound of $n/2$. We can improve this algorithm in a way that will reduce the number of iterations and the average runtime, depending on our data input.

Firstly, we do not need to scan through the entire adjacency matrix. Only half of it will be enough since the adjacency matrix for the undirected graph is diagonally reflective. Also, instead of scanning through the 2d-array by brute-force, solving the modular equation would be faster.

Let *node_val* and *connect_to* be the two vertices from $\{1, \ldots, n\}$ and r be one of residue values. If we want to find *connect_to* for a given *node_val*, r, and m, we can solve the following equation:

$$\begin{aligned} node\_val + connect\_to &= r \pmod m \\ connect\_to &= r - node\_val \pmod m \\ connect\_to &= r - node\_val + m \pmod m \\ connect\_to &= r - node\_val + m + m \pmod m \\ &\vdots \\ connect\_to &= r - node\_val \ (+m + m + \ldots \text{ until } connect\_to \text{ exceeds } n) \end{aligned}$$

The following algorithm is optimized by performing calculation (line 41-57) as above:

```
1  int** adjMatrix(int num_nodes, int mod, vector<int> squareMod, int& num_edges)
       {
2    /** This algorithm calculates the adjacency matrix for the following
3     *  undirected graph:
4     *    - there are 1, 2, ..., num_nodes vertices.
5     *    - connect two vertices if we satisfy the following equation:
6     *        sum of two vertices = square (% mod)
7     *      or  (sum of two vertices % mod) = (square % mod)
```

```
 *      In other words, if (sum of two vertices % mod) is one of the values
 *      in of squareMod, then connect the two vertices via an edge.
 *
 *  @param: int num_nodes, int mod, vector of squareMod (the residue set),
 *        int& num_edges
 *    Note: num_edges is for tracking the number of edges. num_edges has to
 *      be defined before function call and pass it as a function parameter.
 *      Since it is passed by reference, the original num_edges will be
 *      changed as opposed to pass-by-value, which creates a copy of
 *      num_edges and the original num_edges remains unaffected.
 *  @return: pointer to the dynamically allocated 2d array,
 *       which is the adjacency matrix of the graph
 *
 */

// pointer to a dynamically allocated array, which consists of
// pointers to the arrays too. It is an array of the array.
// Thus, this is the 2d array

int** adjMat = new int*[num_nodes];


for (int i = 0; i < num_nodes; i++) { // initialize all entries to 0
  adjMat[i] = new int[num_nodes];
  for (int j = 0; j < num_nodes; j++) {
    adjMat[i][j] = 0;
  }
}

for (int node_val = 1; node_val <= num_nodes; node_val++) {

  for (size_t i = 0; i < squareMod.size(); i++) {

    int connect_to = squareMod[i] - node_val;

    while (connect_to <= node_val) {
      // consider only edges where node_val > connect_to
      // to avoid repetition. For instance, 1--2 edge is
      // also the same as 2--1 edge; hence, redundancy.
      connect_to += mod;
    }

    while(connect_to <= num_nodes) {
      num_edges++;

      adjMat[node_val-1][connect_to-1] = 1; // -1 since index starts at 0
      adjMat[connect_to-1][node_val-1] = 1; // because adjMat is reflective
 diagonally

      connect_to += mod;
    }
```

```
58
59      }
60
61    }
62
63    return adjMat;
64
65 }
66
```

Even though the optimized algorithm does not necessarily change the big-O for worse case runtime, it at least reduces the amount of iterations done. Also, the algorithm hits its worse case only when each residue value produces more than one edge. For instance, when $m = 2$ and $n = 5$, the residue set is just $\{1\}$ and the vertex 1 connects to 2 and 4. On average, it's rare for one residue value to produce more than one edge, at least for our project. Thus, on average, the algorithm has a runtime of $O(n^2)$.

3.3 Finding Hamiltonian Cycle

The naive brute-force approach for finding Hamiltonian cycle would be to enumerate all the possible paths, and we have $n!$ possibilities for n nodes. In terms of constructing a tree, we can start with 1 as a root vertex – in Hamiltonian cycle, the start position does not matter unlike the path – and create children branches for the rest of **unvisited** vertices. Thus, before creating children branches from a particular vertex, we have to keep in track of visited vertices in an array. Then, we have to linearly scan through the visited vertex array before branching out so that we do not create branches that have already been visited. The scan alone takes $O(n)$ worst-case runtime. Hence, the algorithm as a whole takes $O(n \cdot n!)$ runtime. This brute-force algorithm is also mentioned and coded here.

Some optimizations can be done here. In the naive approach, it creates all the branches for unvisited vertices and do not even consider about the connectivity of the graph. In fact, creating a branch for non-existent edge is redundant since we already know there must always exist an edge between any chosen two vertices for a Hamiltonian cycle.

Also, instead of linearly scanning through the visited vertex array every time we want to create an additional branch (which is what $isSafe()$ function from geeksfromgeeks link mentioned above is doing), we can just create a boolean visited vertex array, in which the value at an index i will tell us whether the edge i is already visited or not. In other words, this boolean array behaves just as a characteristic function:

$$f : \{vertices\} \to \{0, 1\}$$

The optimized brute-force algorithm with $O(n!)$ runtime is as follows:

```
1 bool hamiltonian_cycle(int start_node, int** ptr_adjMat, int num_nodes,
2     bool visited[], int edge_count) {
```

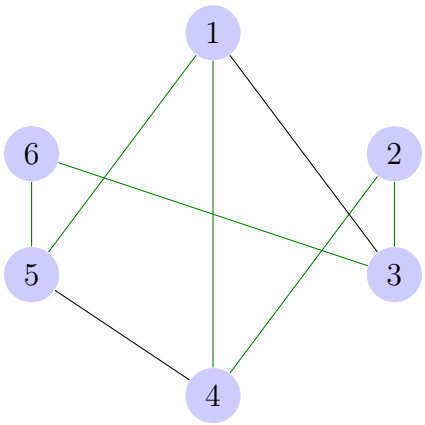
```
3   /** This algorithm checks whether a Hamiltonian cycle is present in a graph
4    *
5    *  @param: the start node, the adjacency matrix of the graph,
6    *       the total number of nodes, an array to track
7    *       visited vertices, edges counter
8    *
9    *  @return: true if Hamiltonian cycle is present; false otherwise
10   *
11   */
12
13  if (edge_count == num_nodes-1 && ptr_adjMat[start_node][0] == 1 && num_nodes
      > 2) {
14    // this is the base case.
15    // 1. The first disjunct is true if we can find a Hamiltonian
16    //    path from starting from start_node.
17    // 2. The second disjunct is true if there is an edge
18    //    connecting the start and end vertices of the path
19    // 3. The third disjunct is for the case with just 1 or 2 vertices
20    //    where Hamiltonian cycle cannot exist by definition
21    return true;
22  }
23
24  visited[start_node] = true;
25
26  for (int col = 0; col < num_nodes; col++) {
27
28    if (visited[col] == 0 && ptr_adjMat[start_node][col] == 1) {
29
30      visited[col] = true;
31
32      if (hamiltonian_cycle(col, ptr_adjMat, num_nodes, visited, edge_count+1)
     ) {
33        return true;
34      }
35
36      // if hamiltonian_cycle is false,
37      // backtracks it
38      visited[col] = false;
39
40    }
41
42  }
43
44  return false;
45
46 }
47
48
```

The for-loop at line 26 tells us that this algorithm searches the branch with lowest value first. In other words, it performs a Depth-First Search (DFS) down the tree. It also does

backtracking when the path from DFS is not a Hamiltonian cycle.

One error that I encountered when I was writing the algorithm was using "++edge_count" and "edge_count++" at line 32, instead of "edge_count+1". The reason is because when we use pre or post increment operators, it also changes the original value of "edge_count"; but when the algorithm does backtracking, the value of "edge_count" should return back to original value. In case that we prefer to use increment operators, one way to get around would be to add "edge_count–" before line 38 (which is before backtracking begins).

As an example, let n (num_nodes) $= 6$ and m (mod value) $= 10$. The following is the data for the graph:




```
Enter num_nodes and mod value, each separated by a space: 6 10

residue for square mod 10: 1 4 9 6 5

Adjacency matrix:
0 0 1 1 1 0
0 0 1 1 0 0
1 1 0 0 0 1
1 1 0 0 1 0
1 0 0 1 0 1
0 0 1 0 1 0

1 conects to 3 4 5 (Degree: 3)
2 conects to 3 4 (Degree: 2)
3 conects to 1 2 6 (Degree: 3)
4 conects to 1 2 5 (Degree: 3)
5 conects to 1 4 6 (Degree: 3)
6 conects to 3 5 (Degree: 2)

num of edges: 8
```

Figure 1: The data of the graph when $m = 10$ and $n = 6$

The tree for the graph looks as follows:

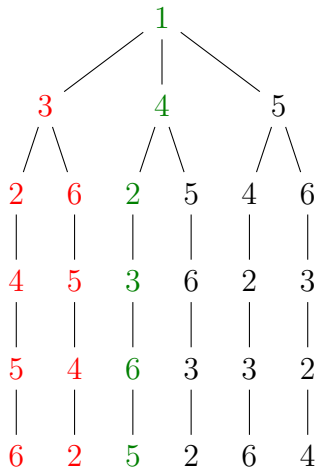


Figure 2: The tree of the graph when $m = 10$ and $n = 6$

The red color vertices are the ones that have gone through backtracking, the green one is the solution that the algorithm has found, the black ones are the one that is not even explored by the algorithm.

The following are the steps that the algorithm is computing:

1. The path generated by the first DFS is: 1–3–2–4–5–6. Since the path contains all the vertices (checked by first conjunct in line 13), we check whether the last vertex, 6 connects to the root vertex, 1. Since it does not, then the path is not a Hamiltonian cycle.

2. Now, the algorithm begins backtracking up the next level from vertex 6 and goes to vertex 5. Since there are no other unexplored branches at vertex 5, it goes up to one additional level. The same happens in vertices 4 and 2 until it reaches vertex 3 when there is an unexplored branch.

   So, the new path found by backtracking and DFS is: 1–3–6–5–4–2. But since 2 does not connect to 1 again, it is not a Hamiltonian cycle.

3. The algorithm undergoes another backtracking and found the path: 1–4–2–3–6–5. This time, 5 connects to 1; so, it forms a Hamiltonian cycle. The algorithm returns true and ends. It does not need to explore any further once it finds **one** Hamiltonian cycle.

4 Results

Before we begin discussing about the results, we want to introduce some new notations to simplify things:

> Let $G_m(n)$ be the graph for the sequence $\{1, 2, \ldots, n\}$ and mod m, and let $R(m)$ be the residue set for mod m.

4.1 Some failures

Upon completion of the program, we were able to easily gather data for multiple values of m and n. Our (with my project parter, Patrick Ward) initial conjecture was:

> If there is a Hamiltonian cycle in the graph $G_m(n)$, then there will always exist a Hamiltonian cycle for $G_k(n)$ where $k \in Z^+$ such that $k > m$.

Our reasoning was that as m increases, $|R(m)|$ also increases and we would have "more options" to connect the vertices. But we found a counter example: $G_2(8)$ have a Hamiltonian cycle while $G_3(8)$ does not.

As a consequence of this failure, we found the following two theorems, which Ward extensively proved in his paper:

> **Theorem 1.** *A Hamiltonian cycle exists in $G_2(n)$ iff n is even*
>
> **Theorem 2.** *The graph $G_3(n)$ does not contain a Hamiltonian cycle or path.*

Proof. of Theorem 1 $\Leftarrow$ direction

Assume n is even. If $m = 2$, $R(m) = \{1\}$.

If the sum of two numbers equal to an odd number, then its mod m is 1.

So, there exists an edge between 1–2, 2–3, 3–4, ..., m–1 (edge from m to 1, not "m minus 1"). Thus, the Hamiltonian cycle is nothing but $1, 2, 3, \ldots, m-1, m, 1$.

□

For more of these mathematical results and proofs, please refer to Ward's paper, *Square-Sum Cycles modulo m*.

4.2 Some interesting patterns on Hamiltonian cycle formation

We found the following results for $m = n$ and n is a prime greater than 9:

1. There always exists a Hamiltonian cycle for primes greater than 9.

   This conjecture is tested for primes from 1 to 1750, using *test_alg_prime*() algorithm. Beyond that range, the program gets "std::bad_alloc" error.

2. The bound at which backtracking will not occur for $G_n(n)$ is $\lceil n/2 \rceil$. This bound *almost always* becomes "looser" as n grows larger.

   In other words, the algorithm performs its usual DFS down the tree and correctly selects $\lceil n/2 \rceil$ of the vertices for guaranteed.

   The following is the data for primes from 11 to 67:

n (no. of vertices)	no. of levels for which no backtracking occurs
11	6 ($\lceil n/2 \rceil$)
13	7 ($\lceil n/2 \rceil$)
17	10 ($\lceil n/2 \rceil + 1$)
19	11 ($\lceil n/2 \rceil + 1$)
23	17 ($\lceil n/2 \rceil + 5$)
29	25 ($\lceil n/2 \rceil + 10$)
31	25 ($\lceil n/2 \rceil + 8$)
37	37 ($\lceil n/2 \rceil + 18$)
41	38 ($\lceil n/2 \rceil + 17$)
43	40 ($\lceil n/2 \rceil + 18$)
47	38 ($\lceil n/2 \rceil + 14$)
53	48 ($\lceil n/2 \rceil + 21$)
59	52 ($\lceil n/2 \rceil + 22$)
61	61 ($\lceil n/2 \rceil + 30$)
67	63 ($\lceil n/2 \rceil + 29$)

   The backtracking process is an expensive path of *hamiltonian_cycle* algorithm. But since we know that backtracking does occur very much, with the bound of $\lceil n/2 \rceil$ levels, our $O(n!)$ is actually faster than we might think for primes n greater than 9 and $m = n$. The next result is the consequence of this finding.

3. Optimized brute-force algorithm for finding Hamiltonian cycle is already fast enough, even for large primes.

   It is tested for primes as large as 22721. Since we know from result (2), our algorithm is fast for the $O(n!)$ runtime. It is also a very slow-scaling algorithm for primes in terms

of runtime. This is because our algorithm is fast when there is a Hamiltonian cycle in the graph, and by result (1), there is always a Hamiltonian cycle for primes greater than 9. On the other hand, when the graph does not have the Hamiltonian cycle, our algorithm will take a very long time because it has to enumerate and check through all the possible Hamiltonian paths starting from vertex 1. Yet, even among the graphs that have Hamiltonian cycle, it is just noticeably faster for primes. For instance, in case of $G_{88}(88)$, it takes a longer time for the algorithm to return an answer for even a small input such as 88, while for $G_{22721}(22721)$, it can return an answer much quicker. We are not able to unearth the mathematical reason behind this phenomenon within a limited time-scope, but nevertheless, it remains such an interesting result to us.

4.3 Further potential research

Many interesting questions were raised and left unanswered during our course of 2-week research, and reflecting on those same questions could help others who plan to further the research on this same topic. The questions are:

1. We know that finding whether there is a Hamiltonian cycle or not in a graph is an NP-complete problem. But our graph has a certain restriction that we can only create an edge between two vertices when the following equation is satisfied:

   $$x + y \equiv a^2 \pmod{m}$$

   where x, y are any two vertices from a given sequence $\{1, 2, \ldots, n\}$ and $a, m \in Z^+$.

   Therefore, given this restriction, can we find a polynomial time algorithm for detecting Hamiltonian cycle? As mentioned before, backtracking is the expensive part of the *hamiltonian_cycle* algorithm, and if we can find a way to foresee when the algorithm will backtrack, then we can make progress to find a faster algorithm? But if there is no Hamiltonian cycle in the graph, can we find a way to determine it in polynomial time?

   If finding a polynomial time algorithm is not possible, can we prove that this problem still belongs in the class of NP (or even NP-complete) problems?

   One approach would be to use machine-learning algorithms to learn from the given data and "predict" when the algorithm will backtrack. But since it is based on prediction, the outcome is always going to be probabilistic. Thus, there is tradeoff between the soundness and speed of the algorithm.

2. Generate mathematical proofs for results from section 4.2.

   In addition, for result (2), can we come up with a "tighter" bound than $\lceil n/2 \rceil$? Is there a pattern or can we come up with a function that takes n as input and return the bound?

For result (3), as we have mentioned, we still do not know the mathematical reason why the algorithm is relatively faster in case of primes. Can we find one? Can we prove it?

3. For the results from section 4.2, we restrict ourselves with the case $n \in \{primes \geq 11\}$ and $m = n$. But if we let m and n be two independent variables, then the runtime and results rather become random. The bound that we found at result (2) will not hold anymore. Can we find any pattern (whatever it is) in this random data?

   One conjecture that I have found so far is that for $k \in Z^+$ such that $k > 16$, $G_k(k)$ always has a cycle. This has not been mathematically proven yet and computationally expensive to generate for large composite k. Note that we are still restricting ourselves to the case $m = n$. Can we mathematically proof whether the conjecture holds or not?

My project partner, Patrick Ward has extensively researched the same topic with emphasis on Mathematical results, theorems, and proofs, rather than a computational one. It would be worthwhile to have a look at his paper, *Square-Sum modulo m*, as well since we had worked together on the same topic, just from different approach.

References

[1] "The Square-Sum Problem. " Performance by Matt Parker, *Numberphile*, 11 Jan. 2018, https://www.youtube.com/watch?v=G1m7goLCJDY

[2] Ward, Patrick *Square-Sum modulo m*, Illinois Wesleyan University

[3] Stangl, Walter D, *Counting Squares in* $\mathbb{Z}_n$. No. 4. Vol. 69, Mathematics Association of America, 1996, 285-289
https://www.maa.org/sites/default/files/Walter_D22068._Stangl.pdf