Assembling Bitcoin

Raymond Feng
Hao-Chun Hsiao
Sophia Shi
Keer Feng
Samuel Lin

Abstract

Bitcoin tackles the double-spending problem by allowing miners to assemble Bitcoin blocks and attach to the end of the blockchain. The blockchain ensures security and the ordered dates of every transaction ever made. The purpose of this paper is to provide an analysis on how the assembly of a Bitcoin block can be modeled by the Independent Set Decision Problem. We have a goal to make the connection between the two problems and explore the complexity classes. In addition, we explore multiple famous algorithms to approximate a solution. Ultimately, coming up with an algorithm of our own to solve the decision problem.

Introduction

Bitcoin is a decentralized currency that does not have a third party monitoring every action made by the users. In order to secure the network from the double-spending problem and reversing transactions, This paper will be focusing on how miners choose to pool their block of transactions. This process can be modeled after an Independent Set Decision Problem. We will prove the polynomial-time reduction by using 3-SAT and the definition of NP-completeness. Furthermore, the paper will explore ways of approximating a solution such as the Greedy Method, Integer Programming, and the Bellman-ford Algorithm. Lastly, we will explain our approach to solving the Independent Set Decision problem.

History of Bitcoin and The Double-Spending Problem

Many consider Bitcoin as the most prominent cryptocurrency in the world. It pioneered a new way on how we view our currencies. It has created a shock wave that caught the attention of billions around the world. Bitcoin has experienced many highs and lows since its first appearance in 2009. However, as of right now each Bitcoin is valued at around 12,000 Canadian dollars. The paper, “Bitcoin: A Peer-to-Peer Electronic Cash System” written by the mysterious Satoshi Nakamoto, unravels the fundamental ideas behind Bitcoin. According to the paper, Bitcoin is “purely peer-to-peer version of electronic cash that would allow online payments to be sent directly from one party to another without going through a financial institution”. In other words, this is a decentralized currency that is not under surveillance by other institutions or governments. The electronic payment system removes the trust of a third financial party and replacing it with a cryptographic proof that ensures security to each transaction made with Bitcoin. As a result, any extra fees to a transaction are minimized and the possibility of reversing a transaction is nearly impossible. This begs the question: How do we trust one and another when a third party does not monitor our actions?

In the world of Bitcoin, each person possesses a personal ledger that holds the record of all the past transactions ever made. Whenever a new transaction is made, the person announces the information to all the bitcoin community with the amount of money being transferred, their own account number and the receiver. Once it is broadcasted, everyone will proceed to update their own ledger. Each account comes with a private key that is linked to the account number. The key allows the user to create digital signatures for each transaction made. The signatures cannot be copied, destroyed or reused in the future, thus, each transaction is unique. While the digital signatures is a strong security blanket that ensures the identity of the person who made the transaction, it is not able to determine the time and date of the transaction. As a result, this can cause a double-spending problem. For example, if a buyer purchases one product each from the supplier (A) and supplier (B). Without the proof of when the transaction was made, both suppliers might think the transaction sent to them is first and proceed to ship their product to the buyer. The networked proposed a solution to the problem that involves the act of “mining”.

According to *Bitcoin: A Peer-to-Peer Electronic Cash System,* The network “hashes each transaction into an ongoing chain of hash-based proof-of-work. The longest chain not only serves as proof of the sequence of events witnessed but proof that it came from the largest proof of CPU power.” The longest chain is also called the Blockchain. The Blockchain consists all past transactions with a complete hash value that allows it to be unique from any other block. Each block stores up to 1 MB of data depending on the size of each transaction sorted into the block. So how do we created these hash values?

As new transactions are made, they go into a pool of transactions in the network that has not been recorded onto the ledgers. Machines or so-called "Miners", will then sort transactions into the right order becoming a block with limited size. The Miners attempt to solve a mathematical problem starting their proof-of-work. The idea is to find a hash code that begins with a number of zero bits. Bitcoin mining uses a hash function called double SHA-256. The hash function takes our block of the transaction and produces a hash value. It is nearly impossible to find a hash value without attempting multiple times with different inputs. However, once the hash value is found it is very easy to check whether or not the hash value is correct. Each block references the one before to ensure the dates are in order which solves the double-spending problem in the network. Currently, the difficulty of finding a correct hash value is no more than one in 10^19. If the miners are able to solve the mathematical problem and the CPU effort can satisfy the proof of work, the Miners will be rewarded with bitcoins and the sorted transaction block is linked to the end of the chain.

When miners decide on which transaction to choose to put into their block we can imagine each transaction as a node in a graph. We will simply connect all the transactions that are with conflict of each other and this will result in an undirected graph. The conflict is exactly the doubling-spending problem. The miner's task is to define whether or not there exists an independent set in the graph. If so, the set can become a block that can go into the hash function.

Independent Set Decision Problem

According to Tutorialspoint, an independent set in graph theory is "represented in sets, in which there should not be any edges adjacent to each other. There should not be any common vertex between any two edges. There should not be ant vertices adjacent to each other. There should not be any common edge between any two vertices".

Given G = (V, E), where V is the number of vertex and E is the number of edges.

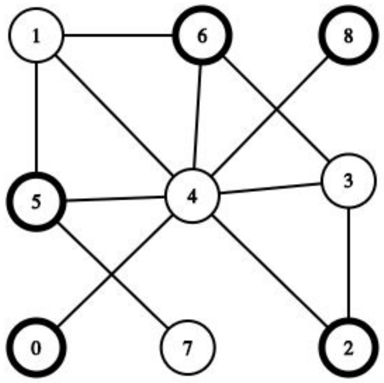


Figure 1

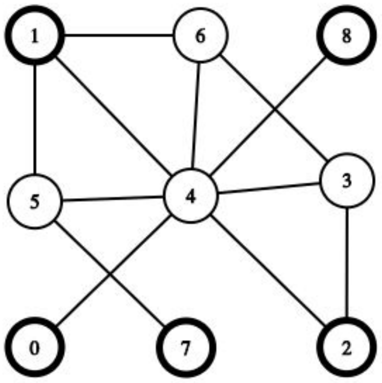


Figure 2

Both figure 1 and figure 2 are the same undirected graph. However, the independent set that is found in can be different based on the vertices we choose. In figure 1, the independent set contains {0 , 2 , 5 , 6 , 8}. In figure 2, the independent set contains {0 , 1 , 2 , 7 , 8} . There are no edges in between any of these vertices, thus, we can conclude that it is an independent set. The Decision version of the problem is to see whether or not a size k independent set exists in a given graph.

Converting Assembling Bitcoin Block to Independent set Decision Problem

Before jumping straight into mathematics, we have to explore the connection between the Bitcoin network and the Independent Set Decision Problem. As mentioned above, the Miners play an important role in ensuring the security of the network. Miners provide a solution by pooling a block of the transaction and a mathematical lottery is held to find the correct hash value. We can mimic the process of assembling a block with an Independent Set Decision Problem. The problem states: given an undirected graph G, is there a subgraph of k vertices with no edges between two vertices? Imagine all transactions as a node and connect an edge between each other when a conflict occurs to form an undirected graph. A set block size is limited to 1MB and the sizes of transactions will alter the number of transactions inside a block, we will assume all transactions to be equal sizes. By dividing the block size and the size of the transaction we can find the amount of transaction in a block. In order words, we have to find the size of the Independent set that we need to check in the decision problem.

NP-completeness

We first explore the complexity classes of NP, NP-hard, and NP-complete. NP (nondeterministic polynomial time) is where the answer to the problem is either “Yes” or “No”. In other words, NP is decision problems. It can be solved in polynomial time and easily verified. NP-Hard problems that are at least as hard as the hardest problems in NP. According to the definition, “A decision problem H is NP-hard when for every problem L in NP, there is a polynomial-time reduction from L to H.” An NP-complete problem is when a problem is in NP and also in NP-hard. To prove the Np-completeness of the Independent set decision problem we have to show that is in NP and is NP-Hard.

For each vertex that is chosen in an independent set, we can check if it has an edge connecting to another vertex in the set. If there exists an edge between any two vertices then the set is rejected. However, we will accept the set when no edge is found. This is an easy way to check and see if our answer is correct or not. Thus, this satisfies the definition of NP, so the Independent Set Decision Problem is in NP.

To show that it is NP-hard, we will give a reduction from 3-SAT to the Independent set problem. We will give an instance of 3-SAT and produce a graph G(V, E) such that G has one or more independent set if and only if the 3-SAT Boolean formula is satisfied. A 3-SAT formula consists of literals that are either a variable or its negation. Literals can only be true or false. Disjunctions between three literals form a *clause*. Each clause in a conjunctive form with other clauses can form a 3-SAT formula. For example,

$$(P \vee Q \vee R) \wedge (P \vee \sim Q \vee R) \wedge (P \vee \sim Q \vee \sim R)$$

In order for the above 3-SAT formula to be true, at least one literal in each clause has to be true. Thus, we will connect all the vertices within each clause cluster to each other, forcing us to choose exactly one literal from each clause. In addition, we have to make sure that a variable and its negation cannot be chosen into the same independent set. Thus, we connect edges between them which are also called the *conflict link.* In terms of Bitcoin, we can assume that the conflict links are the transactions that are in conflict with one another.

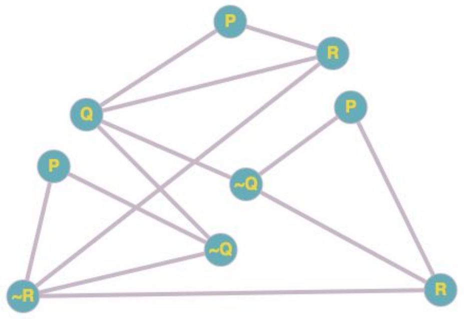


Figure 3

Figure 3 is this completed graph after following all the rules of 3-SAT. By looking at this undirected graph, we can conclude that there exists an independent set such as {P, ~Q, ~R} or {R, P, ~Q}. We have now successfully reduced 3-SAT to an Independent Set Decision Problem. Thus, the problem is NP-Hard and the NP-completeness is proven.

Co-NP

Focus on the computational complexity, we made our research topic into a decision problem. Thus, the output will only give “Yes” or “No”. we assume there is an independent set of size k in G(V, E). On the one hand, the output will be “Yes” in NP. On the other hand, the output will be “No” in Co-NP. Thus, checking the given answers are correct or not is computable. For example, if let k = 3 and there exists an Independent set of size k in NP. The question for Co-NP will produce “No”. In other words, “No, there is an independent set of size k”. We can check by finding an Independent set of size 3 in the graph. To summarize, the NP and Co-NP statements are listed below:

(a).Co-NP: Is there not an independent set of size k in G(V, E).

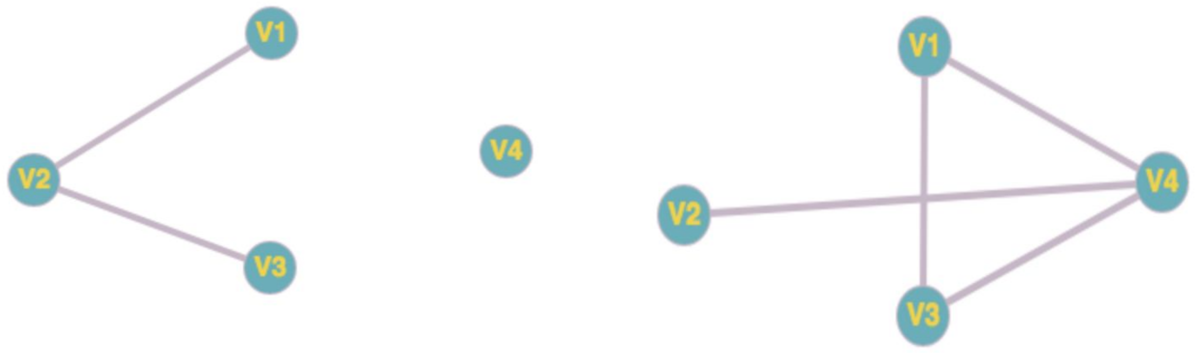
Co-NP Statement:All independent set |G(V,E)| < K in the given graph.

(b).NP: Is there an Independent Set of size k in G(V,E)

NP-Statement:find Independent Set in |G(V,E)|>= k

Clique Reduction to Independent Set Problem

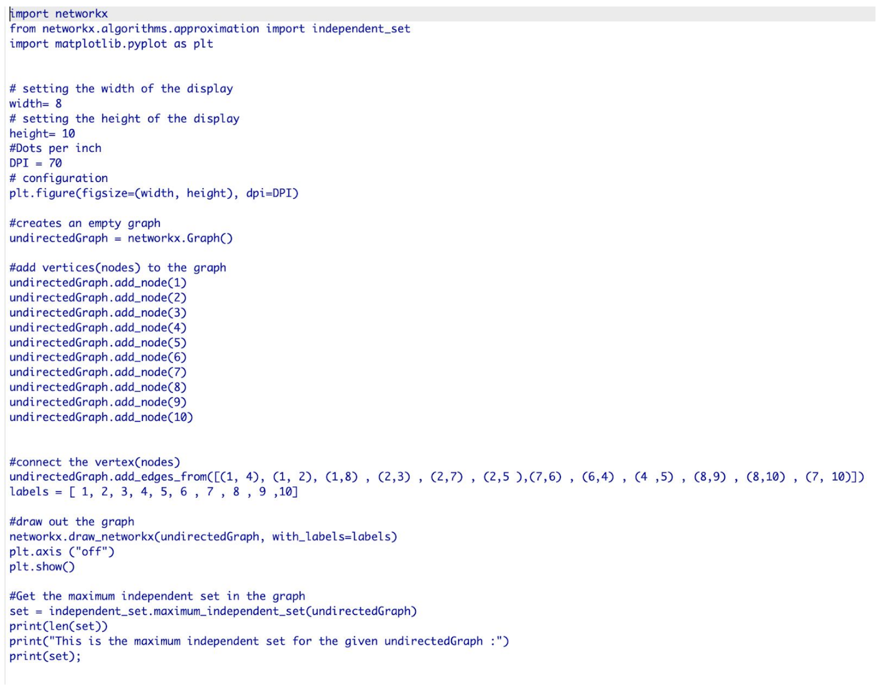
Clique reduction can lead to an independent set that can show NP-Completeness. Thus, it is also an NP-hard problem. As mentioned above, NP-hard is a problem that is at least as hard as the hardest



problems in NP. Imagine, a graph G, a clique in the graph is a subset of vertices of an undirected graph such that every two distinct vertices in the clique are adjacent and the complement graph of a graph *G* is a graph *H* on the same vertices such that two distinct vertices of *H* are adjacent if and only if they are not adjacent in *G*. Referring to the graph below, we can easily see that the independent set of the graph on the left are the vertices that did not connect to one another, which are vertices V1 V2 and V3. And the graph on the right shows the complementary graph of it. By definition, the clique will be the complete graph forming by V1 V3 and V4.

The clique reduction problem takes a graph G=(V, E) and an integer *k*, and asks if G contains a clique of size k. By constructing a complementary graph G'=(V',E') such that V=V' will have the same vertices as the original graph and G' has all the edges that's not present in G. Check if there's an independent set of size k in the complementary graph G'(no two vertices share an edge in G'), then there exists a clique of size k. As we mentioned from the last section we have proven that reduction from 3-SAT to the Independent set problem can show that it's an NP-hard. We couldn't find a direct solution to prove that the clique is NP-hard, but by the reduction from 3-SAT, and the research we did show that the clique problem has at least equivalent hardness as the IS. Since we have proven that IS is an NP-hard problem, which implies that Clique reduction is also an NP-hard problem.

Using python and the packages such as NetworkX and matplotlib we are able to program the approximation of Maximum Independent Problem. The program is able to display the graph that is being constructed. We pass the undirected Graph into a function from NetworkX called maximum_independent_set(). The function will then return a maximum independent set for the given undirected Graph. According to NetworkX, the approximation uses the framework of the "subgraph-excluding" algorithm and the time complexity is O(n/(logn)^2).



```
import networkx
from networkx.algorithms.approximation import independent_set
import matplotlib.pyplot as plt


# setting the width of the display
width= 8
# setting the height of the display
height= 10
#Dots per inch
DPI = 70
# configuration
plt.figure(figsize=(width, height), dpi=DPI)

#creates an empty graph
undirectedGraph = networkx.Graph()

#add vertices(nodes) to the graph
undirectedGraph.add_node(1)
undirectedGraph.add_node(2)
undirectedGraph.add_node(3)
undirectedGraph.add_node(4)
undirectedGraph.add_node(5)
undirectedGraph.add_node(6)
undirectedGraph.add_node(7)
undirectedGraph.add_node(8)
undirectedGraph.add_node(9)
undirectedGraph.add_node(10)


#connect the vertex(nodes)
undirectedGraph.add_edges_from([(1, 4), (1, 2), (1,8) , (2,3) , (2,7) , (2,5 ),(7,6) , (6,4) , (4 ,5) , (8,9) , (8,10) , (7, 10)])
labels = [ 1, 2, 3, 4, 5, 6 , 7 , 8 , 9 ,10]

#draw out the graph
networkx.draw_networkx(undirectedGraph, with_labels=labels)
plt.axis ("off")
plt.show()

#Get the maximum independent set in the graph
set = independent_set.maximum_independent_set(undirectedGraph)
print(len(set))
print("This is the maximum independent set for the given undirectedGraph :")
print(set);
```

Greedy approximation

In scheduling some specific questions while facing the problem that it is inherently difficult to get access to the answer, the greedy method is a well-known algorithm to approximate the maximum independent problem by building a solution step by step.

The maximum degree of the graph is $\Delta(G)$. The neighbor vertices set containing v is the subset of vertices near v and denoted as $N_G(V)$. The subgraph of G is from a subset $W \subseteq V$ is denoted by G[W].

The Greedy algorithm is defined by the procedure as follows, and the output S is a greedy set with the elements in sets are picked nodes in the graph. When the V and E are not explicit, we denote the vertices as V(G) and the edges as E(G):

```
Require: a graph G=(V, E)
  $W \leftarrow V$
  $S \leftarrow \varnothing$
  while W $\varnothing$ do
  Find a vertex $v \in W$ with a minimum degree in G[W]
  $W \leftarrow W \setminus N_G[v]$
  $S \leftarrow S \cup \{v\}$
  end while
return S
```

By using the procedure above, we can easily check if the greedy set is a maximum independent set.

As we proved before, the maximum independent set is an NP-Hard problem, which generally indicates a set of unconnected two vertices with no edges in the conflict graph. It is impossible to find a non-trivial polynomial-time solution for maximum independent algorithms so that the greedy approximation will be utilized in this problem. The greedy algorithm consists of a set of optimal moves to approach the solution. These methods are usually quick and do not need complicated space algorithm.

For the general maximum independent set graph, the basic idea is to select a vertex in each step putting in the solution. For example, there are three sets of nodes, the starting point is marked as *x*, and then we need to find all its neighbors, with the first set of nodes called *A1, A2, A3, …, Ak.* Then we remove this vertex and all its neighbors from the graph, with no connected vertices remaining. In each step, we wish to find the best vertex to get the best independent set in the end. The fewer vertices we removed, the better solution we will get finally which means it seems to find out the vertex in every step with the minimum degree in all nodes. Although in the worst situation, the greedy algorithm is the best way to approximate the maximum independent set because it can approach the n-approximate ratio, while the n represents the number of the nodes.

The greedy approximation works well in a random graph and it is also important to know when it fails and successes.

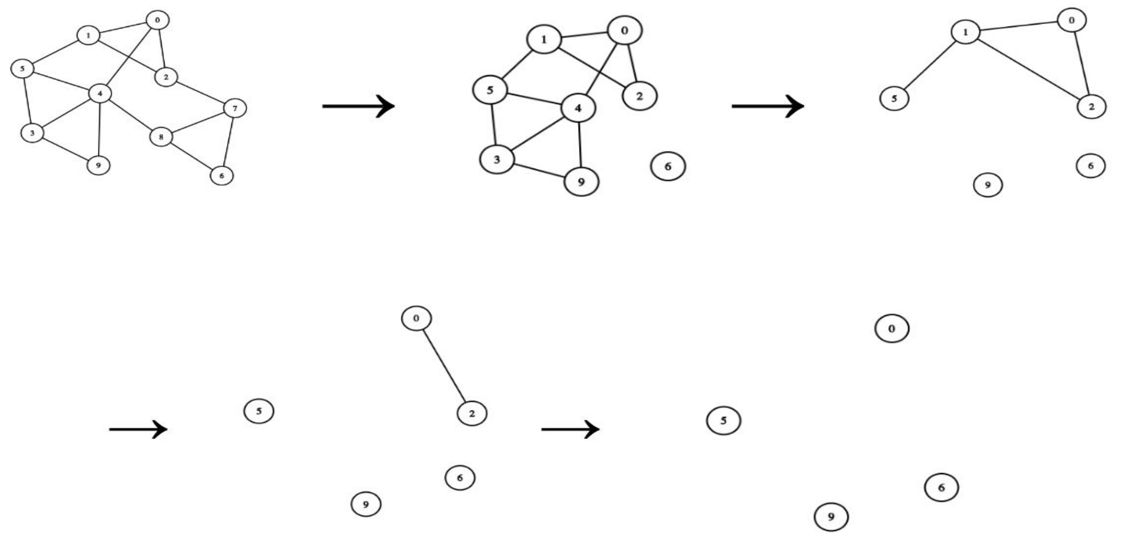
As the example we show before, we select the vertex *x*, with the connection of each Ai and Ai connects to each *Bj;* and *Bj* forms a clique. In this graph, x chooses a node in *B1, B2, B3, …, Bk* and in the end stops. While the independent set will return with the size of 2, the approximation ratio is nearly the same with the size of the instance. In this situation, the result of solving the maximum independent set using the greedy algorithm is failed.

If we consider the bounded graph rather than high degree nodes, then for the greedy degree, Δ, will achieve the approximation ratio $\frac{\Delta+2}{3}$ which is not that bad. The approximation ratio we get can be improved for some subclasses. Here is an example that the approximation ratio will get infinity when n grows.

Let S be a greedy set on a graph G with n vertices and average degree d. We will get $|S| \geq \frac{n}{d+1}$ which shows a better approximation for the maximum independent set problem and is better than $\frac{\Delta+2}{3}$. We can prove this result that whenever the vertex v picked by Greedy, there are at most $\Delta(v)$ are removed and there are at most $|S|\Delta(G)$ vertices in the end removed. Therefore, G has at most $|S|+|S|\Delta(G)$ vertices.

For the lower degree graph, there are some more accurate bounds, like $\frac{5}{3}$ for graphs with the degree at most 3 and $\frac{3}{2}$ in cubic graphs.

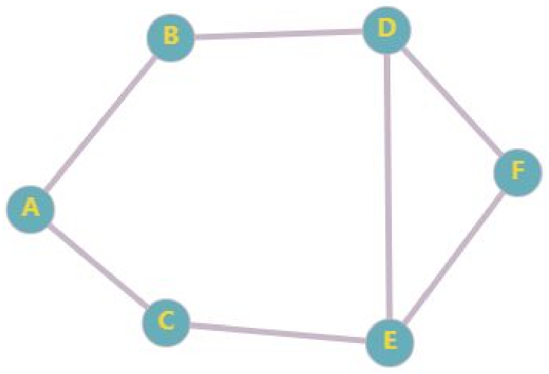
Greedy Approximation Example



To visualize the Greedy approximation algorithm, we first set up an undirected graph with 9 vertices then we select the vertex with the least degree and remove the connecting vertices. 2 degrees are the least degrees connecting vertices in the first graph which are vertex 6 and 9. We select vertex 6 and remove the vertex of 7 and 8 which leads us to graph 2. We then selected the least degree vertex from graph 2. Select vertex 9 and remove vertex 3 and 4 as shown in Graph 3. We do the same thing for every graph until there are no more edges remaining, which we can see on the last graph, there's vertex 0, 5 , 6 and 9 remaining. This forms an Independent set.

Example of Greedy Approximation when it fails

The Greedy algorithm is not ideal, thus there is a chance the out is not the optimal solution. The following graph is the smallest graph that the Greedy algorithm can fail.



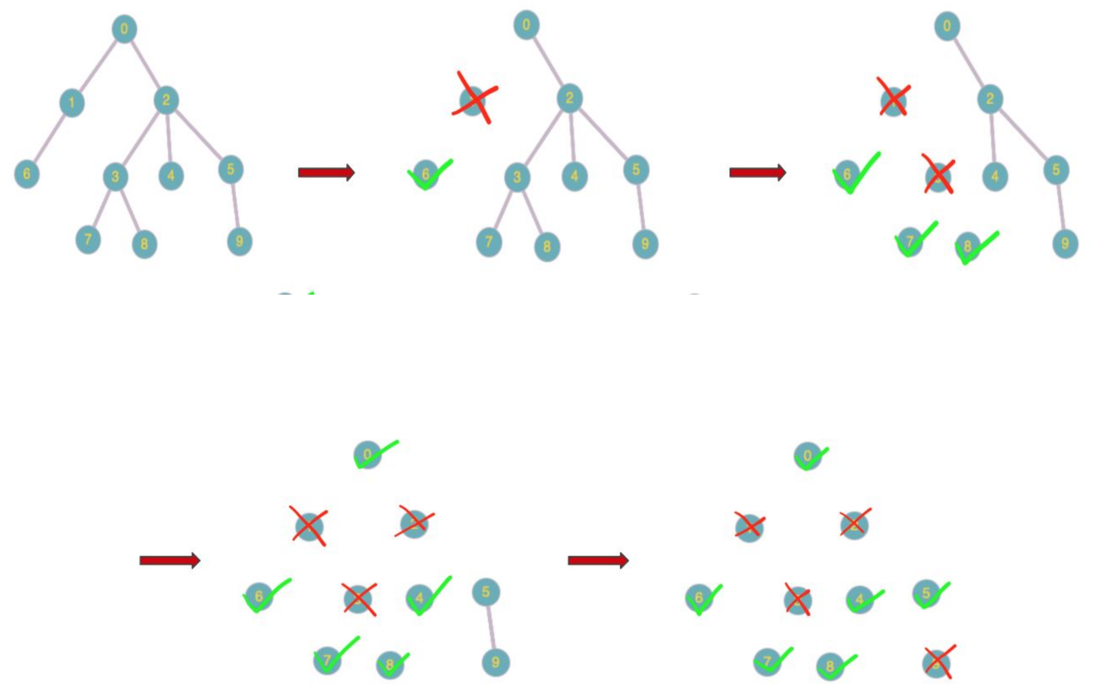
When we pick the least degree connecting vertices, if we chose vertex A for the first selection we will discard vertex b and c. For the next selection only one of the remaining three Vertices D, E and F will be selected. Thus, the size of the independent set is 2. However, if we start the Greedy approximation with Vertex F, the result will lead to Vertex F, A and one of Vertex B or C, which is a size 3 solution. Therefore we can conclude that Greedy approximation does not give an optimal solution in some cases. The ways we select the least connecting vertex will result in different solutions.

Tree Graphs

A tree is an undirected graph in which any two vertices are connected by exactly one path. In the maximum independent set problem, a tree graph is a special case where we can find the optimal solution by applying the greedy algorithm.

Consider the tree graph below 6, 7, 8, 4 and 9 all have only one neighbor, select one of them, 6 in this case, we mark 6 as selected and discard it's neighbor 1 and all its edges. Pick the next least connected vertex and mark it as selected, 7 in this case, discard it's neighbor 3 and all its edges. Now 8 has no neighbors which makes it the least connected vertex, so we will mark it as selected.

Repeat the process, mark 0 discard 2 and mark 4. Finally, pick one from 5 or 9 and we will have our independent set of size 6.



Integer Linear programming

Integer linear programming is a handy way to solve optimization and feasibility problems, and the logic structure makes it easy to program.

In order to find an integer linear programming formulation to the independent set problem of a given graph G, we introduced a decision variable $x_i \in \{0,1\}$ to every vertex I in G, where $x_i = 1$ if i is included in the independent set and $x_i = 0$ if i is not included. Next, we set a constraint $x_i + x_j \leq 1$ for every edge $\{x_i , x_j\}$ in G. Finally the objective is to assign a value either 0 or 1 to every variable x so that the sum of all x's is the greatest.

Consider figure 1, the ILP formulation would be:

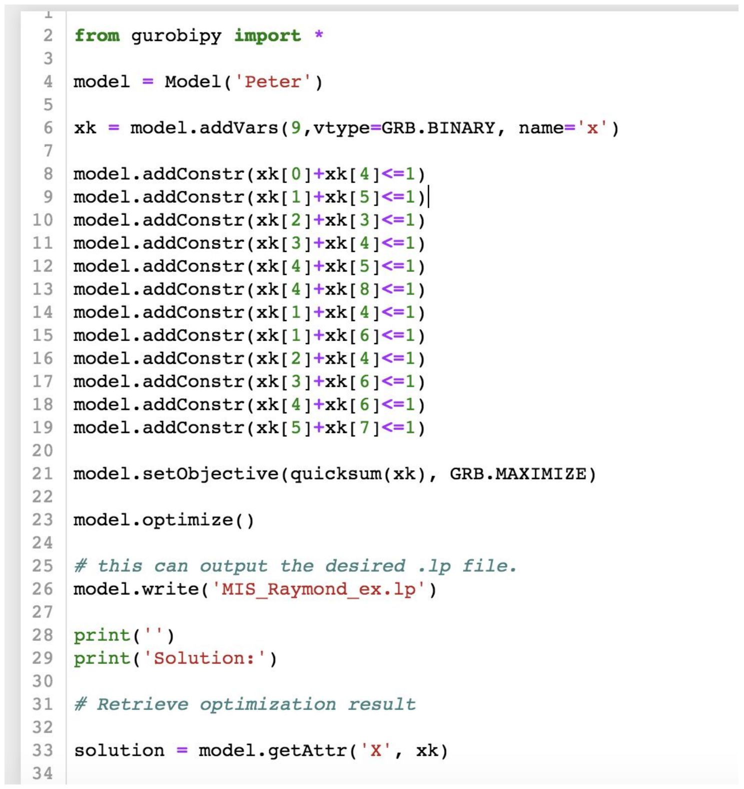
Maximize $\quad x_0 + x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8$

$$\begin{aligned}
&x_0 + x_4 \leq 1\\
&x_1 + x_4 \leq 1\\
&x_1 + x_5 \leq 1\\
&x_1 + x_6 \leq 1\\
&x_2 + x_3 \leq 1\\
&x_2 + x_4 \leq 1\\
&x_3 + x_4 \leq 1\\
&x_3 + x_6 \leq 1\\
&x_4 + x_5 \leq 1\\
&x_4 + x_6 \leq 1\\
&x_4 + x_8 \leq 1\\
&x_5 + x_7 \leq 1
\end{aligned}$$

$x_0 , x_1 , x_2 , x_3 , x_4 , x_5 , x_6 , x_7 , x_8 \in \{0,1\}$

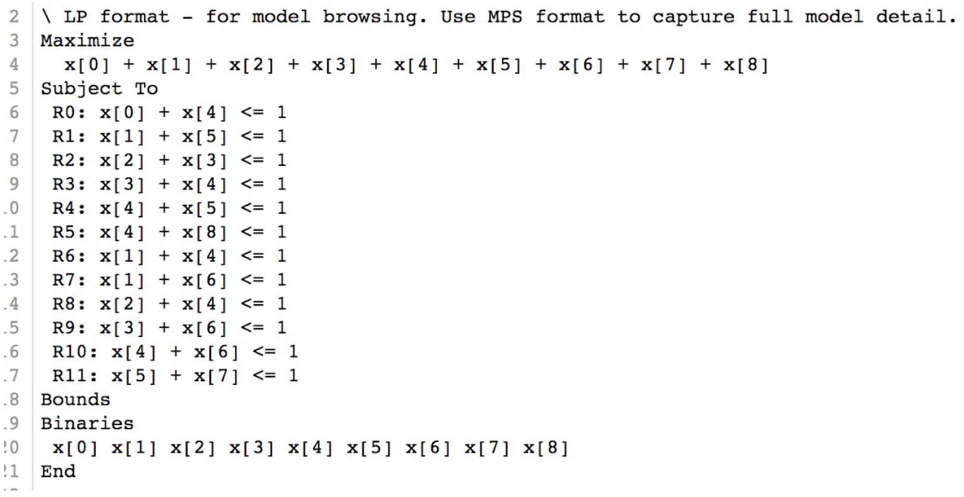
Python Code:

We use the Python library - "gurobi " as the optimization tool in our research. Firstly, set up the target function with nine variables and name the target variable as "xk" in code. Secondly, construct the constraints on the nodes. To write down every connected vertex within the given constraint less or equal to 1.



```python
from gurobipy import *

model = Model('Peter')

xk = model.addVars(9,vtype=GRB.BINARY, name='x')

model.addConstr(xk[0]+xk[4]<=1)
model.addConstr(xk[1]+xk[5]<=1)
model.addConstr(xk[2]+xk[3]<=1)
model.addConstr(xk[3]+xk[4]<=1)
model.addConstr(xk[4]+xk[5]<=1)
model.addConstr(xk[4]+xk[8]<=1)
model.addConstr(xk[1]+xk[4]<=1)
model.addConstr(xk[1]+xk[6]<=1)
model.addConstr(xk[2]+xk[4]<=1)
model.addConstr(xk[3]+xk[6]<=1)
model.addConstr(xk[4]+xk[6]<=1)
model.addConstr(xk[5]+xk[7]<=1)

model.setObjective(quicksum(xk), GRB.MAXIMIZE)

model.optimize()

# this can output the desired .lp file.
model.write('MIS_Raymond_ex.lp')

print('')
print('Solution:')

# Retrieve optimization result

solution = model.getAttr('X', xk)

```

Output:



```
\ LP format - for model browsing. Use MPS format to capture full model detail.
Maximize
  x[0] + x[1] + x[2] + x[3] + x[4] + x[5] + x[6] + x[7] + x[8]
Subject To
 R0: x[0] + x[4] <= 1
 R1: x[1] + x[5] <= 1
 R2: x[2] + x[3] <= 1
 R3: x[3] + x[4] <= 1
 R4: x[4] + x[5] <= 1
 R5: x[4] + x[8] <= 1
 R6: x[1] + x[4] <= 1
 R7: x[1] + x[6] <= 1
 R8: x[2] + x[4] <= 1
 R9: x[3] + x[6] <= 1
 R10: x[4] + x[6] <= 1
 R11: x[5] + x[7] <= 1
Bounds
Binaries
 x[0] x[1] x[2] x[3] x[4] x[5] x[6] x[7] x[8]
End
```

```
Optimize a model with 12 rows, 9 columns and 24 nonzeros
Variable types: 0 continuous, 9 integer (9 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [1e+00, 1e+00]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+00]
Found heuristic solution: objective 5.0000000
Presolve removed 12 rows and 9 columns
Presolve time: 0.01s
Presolve: All rows and columns removed

Explored 0 nodes (0 simplex iterations) in 0.01 seconds
Thread count was 1 (of 4 available processors)

Solution count 1: 5

Optimal solution found (tolerance 1.00e-04)
Best objective 5.000000000000e+00, best bound 5.000000000000e+00, gap 0.0000%

Solution:
{0: 1.0, 1: 1.0, 2: 1.0, 3: 0.0, 4: 0.0, 5: 0.0, 6: 0.0, 7: 1.0, 8: 1.0}
```

LP Rounding

As one of Karp's 21 NP-complete problems, integer linear programming is also hard to solve. Thus, we have to introduce a rounding method from linear programming in order to solve the problem in polynomial time.

First we have to ignore the integer constraint and transform our ILP into LP with $x_i \in [0,1]$. To find an integer solution set, we round up all $x_i > 0.5$ to 1 and the rest to 0. Because for each edge $\{x_i , x_j\}$, x_i and x_j can not be both greater than 0.5, they will not be both rounds up to 1, hence we now have a solution to our independent set problem. Since linear programming is in P and this rounding algorithm can be done in O(n) time, the solution we obtained is a polynomial-time approximation.

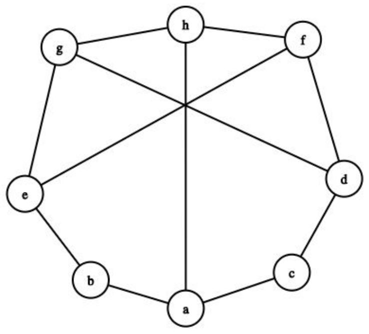
Looking back at the output provided we found a solution to the linear programming formulation which is

$$x_0 = 1 \quad x_1 = 1 \quad x_2 = 1 \quad x_3 = 0 \quad x_4 = 0 \quad x_5 = 0 \quad x_6 = 0 \quad x_7 = 1 \quad x_8 = 1$$

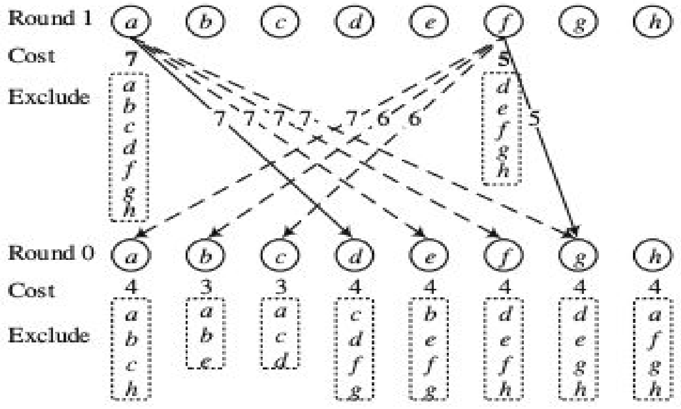
Since all values are integers, the set [0 , 1 , 2 , 7 ,8] is the maximum independent set.

Bellman-Ford Algorithm

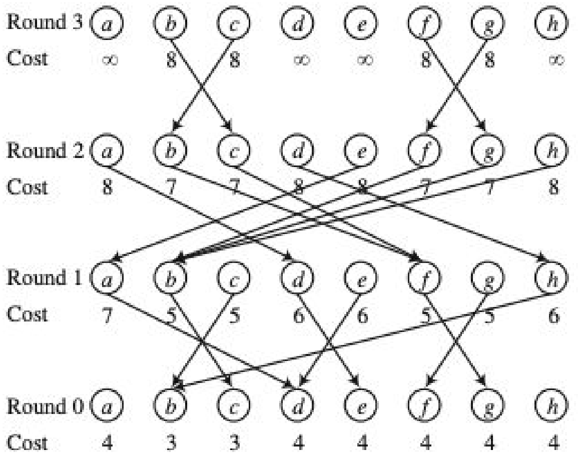
According to the paper *Maximum Independent Set Approximation Based on Bellman-Ford Algorithm* By Mostafa H. Dahshan, "The goal of the algorithm is to minimize the number of excluded vertices for each vertex added to the path. The algorithm works in multiple rounds. To keep track of all paths, the algorithm maintains the lists *Exclude* , *Cost,* and *Previous* for each round. BMA algorithm can run up to |v| -1 rounds. Each round is a new attempt to find a higher independence number. It will stop at the round where no more vertices can be added to any path(independent set)". To simply put it, the *Exclude* of a vertex is the set of vertices that are connected to the vertex including itself. The *cost* is the cardinality of the Exclude and each round will refer back to the *Previous* round of *Exclude* to come up with a new set of Exclude from the next round until no more vertices can be added. Given the graph below:



There are 8 vertices in the graph, to find the Independent set we first have to set up the Exclude and the cost of each vertex. For example, vertex a will have the exclude of [b, a, c, h] and the cost of 4. We do this for every vertex in round 0. We will get:



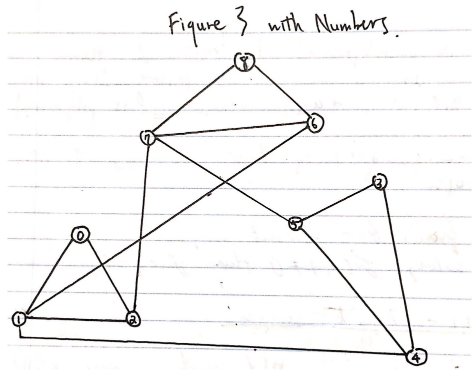
In order to move on to the next round, we will have each vertex point towards the vertices that are not connected to them. In the diagram above vertex a is pointing towards d , e , f and g , vertex b is pointing towards a , b , c , and g. We then look at the union of its own Exclude and the non-connect Exclude and choose the one with the lowest cardinality to become the next round of Exclude. We do this for all vertices. The last vertices to complete their Exclude with all vertices we can trace back the arrows and find the Independent set. The diagram below is the complete rounds of the graph given.

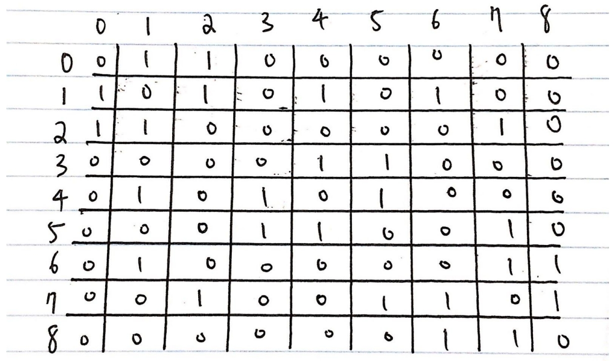


As we can see, vertex b , c , f and g are the last ones to complete their Exclude we can trace back the arrows. For example, we can start at vertex b in round 3 and following the arrows we go to vertex c at round 2 then we go to vertex f at round 1, lastly, we go to vertex g at round 0. As a result we sort the vertices that we passed through which is [b , c , f, g] and this will form an Independent Set. The time complexity of this algorithm is O(n(n^2 -m)).

Our Solution

To find a solution to the NP-complete problem we have to come up with an algorithm. For the solution, we decided to use Java to code our solution method. We have to model an undirected graph to a matrix of zeros and ones so that the computer would understand it. Suppose we have a Graph G = (V, E), the dimension of the matrix is the number of vertices in a graph. Thus, dimension = V x V . We number the vertices. For all the vertices that have an edge between them we will set it to 1 and the vertices that do not have an edge we will set it to 0. For example, referring back to figure 3, there 9 vertices in total and we can number the vertices from 0 to 8. We can model the matrix like this:





	0	1	2	3	4	5	6	7	8
0	0	1	1	0	0	0	0	0	0
1	1	0	1	0	1	0	1	0	0
2	1	1	0	0	0	0	0	1	0
3	0	0	0	0	1	1	0	0	0
4	0	1	0	1	0	1	0	0	0
5	0	0	0	1	1	0	0	1	0
6	0	1	0	0	0	0	0	1	1
7	0	0	1	0	0	1	1	0	1
8	0	0	0	0	0	0	1	1	0

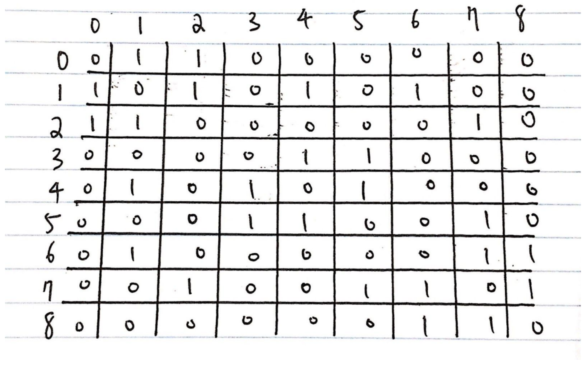
Once we have the matrix we can develop the algorithm. Our task is to find a size k Independent Set in the given graph. Instead of going through all the possible combinations of size k we can reduce it. As we can see in the matrix, the value of the rows and columns are the same. The rows of the matrix above are already labeled with numbers. Thus, we can choose to let each row to represent each node. For each row, we will remove the nodes that have an edge between the selected node. Take node 0 as an example, we see that it has edges between vertex 1 and 2, we can remove those two nodes leaving us with the set of nodes [0, 3, 4, 5, 6, 7, 8]. We do this for each row. We will get :

Vertex 0: [0 , 3 , 4 , 5 , 6 , 7 , 8]   Vertex 1: [1 , 2 , 4 , 6]   Vertex 2: [2 , 3 , 4 , 5 , 6 , 8]
Vertex 3: [0 , 1 , 2 , 3 , 6 , 7 , 8]   Vertex 4: [0 , 2 , 4 , 6 , 7 , 8]   Vertex 5: [0 , 1 , 2 , 5 , 6 , 8]
Vertex 6: [0 , 2 , 3 , 4 , 5 , 6]   Vertex 7: [0 , 1 , 3 , 4 , 4]   Vertex 8: [0 , 1 , 2 , 3 , 4 , 5 , 8]

Each set of numbers contains the vertex N and the numbers(other vertices) are not adjacent to the vertex N. For each set we would have to find all combinations of size k and check if it is independent or not. Let k = 3, and we take the set for vertex 0. The combinations formula would be 7!/(3! * (7 - 3)!) = 35. There would be 35 combinations of size 3 from the set from vertex 0 :

[0, 3, 4], [0, 3, 5], [0, 3, 6], [0, 3, 7], [0, 3, 8],
[0, 4, 5], [0, 4, 6], [0, 4, 7], [0, 4, 8], [0, 5, 6],
[0, 5, 7], [0, 5, 8], [0, 6, 7], [0, 6, 8], [0, 7, 8],
[3, 4, 5], [3, 4, 6], [3, 4, 7], [3, 4, 8], [3, 5, 6],
[3, 5, 7], [3, 5, 8], [3, 6, 7], [3, 6, 8], [3, 7, 8],
[4, 5, 6], [4, 5, 7], [4, 5, 8], [4, 6, 7], [4, 6, 8],
[4, 7, 8], [5, 6, 7], [5, 6, 8], [5, 7, 8], [6, 7, 8]

To check that each set we have to refer back to the matrix of ones and zeros. We take set [0,3,4] and [0, 3 ,8]. For [0,3,4], we will start at row 0 and see if column 3 and 4 is either a 1 or a 0. In this case, they are both 0s which

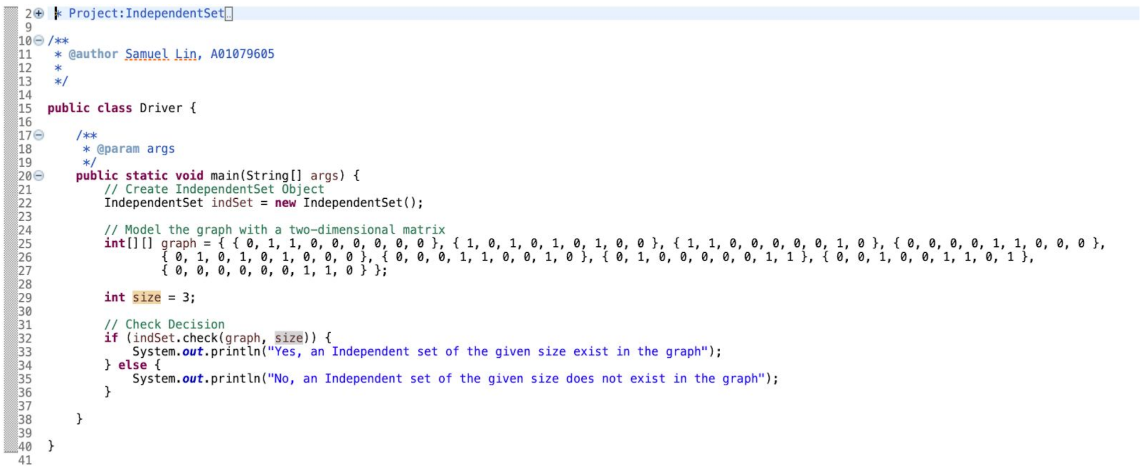


	0	1	2	3	4	5	6	7	8
0	0	1	1	0	0	0	0	0	0
1	1	0	1	0	1	0	1	0	0
2	1	1	0	0	0	0	0	1	0
3	0	0	0	0	1	1	0	0	0
4	0	1	0	1	0	1	0	0	0
5	0	0	0	1	1	0	0	1	0
6	0	1	0	0	0	0	0	1	1
7	0	0	1	0	0	1	1	0	1
8	0	0	0	0	0	0	1	1	0

means that there are no edges between them. We will go to the next digit in the set. In row 3, we will check columns 0 and 4. At the row-column 0 contains a value of 0 and column 4 contains a value of 1. The validation will stop because there exists an edge between vertex 3 and 4 showing that the set is not independent. For the set [0 , 3, 8], we start from row 0 and check column 3 and 8. We will see that all the values are 0 We do this for all three rows and will find that all values are 0s. Thus, we can conclude that the set is independent. Link to the GitHub page of the source code:
https://github.com/LinsamuelATM/Math441-Assembling-Bitcoin-Solution

Main class:

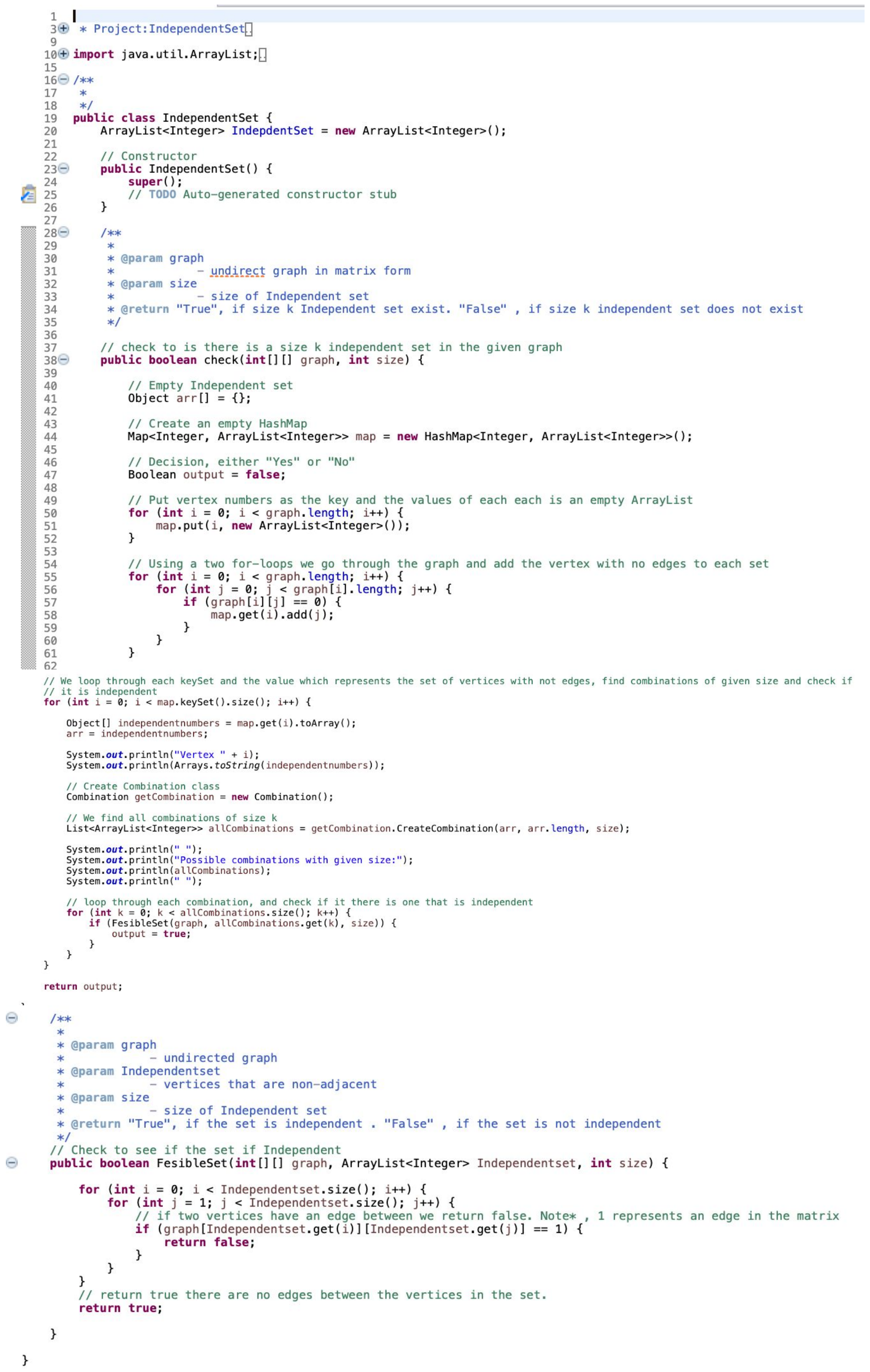
In this class we will set up the matrix to represent an undirected graph using a two-dimensional array.



```
 * Project:IndependentSet

/**
 * @author Samuel Lin, A01079605
 *
 */

public class Driver {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // Create IndependentSet Object
        IndependentSet indSet = new IndependentSet();

        // Model the graph with a two-dimensional matrix
        int[][] graph = { { 0, 1, 1, 0, 0, 0, 0, 0, 0 }, { 1, 0, 1, 0, 1, 0, 1, 0, 0 }, { 1, 1, 0, 0, 0, 0, 0, 1, 0 }, { 0, 0, 0, 0, 1, 1, 0, 0, 0 },
                { 0, 1, 0, 1, 0, 1, 0, 0, 0 }, { 0, 0, 0, 1, 1, 0, 0, 1, 0 }, { 0, 1, 0, 0, 0, 0, 0, 1, 1 }, { 0, 0, 1, 0, 0, 1, 1, 0, 1 },
                { 0, 0, 0, 0, 0, 0, 1, 1, 0 } };

        int size = 3;

        // Check Decision
        if (indSet.check(graph, size)) {
            System.out.println("Yes, an Independent set of the given size exist in the graph");
        } else {
            System.out.println("No, an Independent set of the given size does not exist in the graph");
        }

    }

}
```

IndependentSet class:

The check() function takes in a graph and the size that it will check. The function starts off by creating an empty array and initializing a Hashmap. The first for-loop creates an empty set for each vertex for every key value of the map. We then use a double for-loop to loop through the two-dimensional array and find the rows and columns with the value of 0 to add to each set. The key values of the map represent each vertex. For each key-value we get the collections of numbers and find all combinations of size k. We then use a for-loop and called FesibleSet() and check whether or if any of the combinations are independent. Finally, the check() function returns a boolean value.



```java

 * Project:IndependentSet

import java.util.ArrayList;

/**
 *
 */
public class IndependentSet {
    ArrayList<Integer> IndepdentSet = new ArrayList<Integer>();

    // Constructor
    public IndependentSet() {
        super();
        // TODO Auto-generated constructor stub
    }

    /**
     *
     * @param graph
     *            - undirect graph in matrix form
     * @param size
     *            - size of Independent set
     * @return "True", if size k Independent set exist. "False" , if size k independent set does not exist
     */

    // check to is there is a size k independent set in the given graph
    public boolean check(int[][] graph, int size) {

        // Empty Independent set
        Object arr[] = {};

        // Create an empty HashMap
        Map<Integer, ArrayList<Integer>> map = new HashMap<Integer, ArrayList<Integer>>();

        // Decision, either "Yes" or "No"
        Boolean output = false;

        // Put vertex numbers as the key and the values of each each is an empty ArrayList
        for (int i = 0; i < graph.length; i++) {
            map.put(i, new ArrayList<Integer>());
        }

        // Using a two for-loops we go through the graph and add the vertex with no edges to each set
        for (int i = 0; i < graph.length; i++) {
            for (int j = 0; j < graph[i].length; j++) {
                if (graph[i][j] == 0) {
                    map.get(i).add(j);
                }
            }
        }

        // We loop through each keySet and the value which represents the set of vertices with not edges, find combinations of given size and check if
        // it is independent
        for (int i = 0; i < map.keySet().size(); i++) {

            Object[] independentnumbers = map.get(i).toArray();
            arr = independentnumbers;

            System.out.println("Vertex " + i);
            System.out.println(Arrays.toString(independentnumbers));

            // Create Combination class
            Combination getCombination = new Combination();

            // We find all combinations of size k
            List<ArrayList<Integer>> allCombinations = getCombination.CreateCombination(arr, arr.length, size);

            System.out.println(" ");
            System.out.println("Possible combinations with given size:");
            System.out.println(allCombinations);
            System.out.println(" ");

            // loop through each combination, and check if it there is one that is independent
            for (int k = 0; k < allCombinations.size(); k++) {
                if (FesibleSet(graph, allCombinations.get(k), size)) {
                    output = true;
                }
            }
        }

        return output;
`
    /**
     *
     * @param graph
     *            - undirected graph
     * @param Independentset
     *            - vertices that are non-adjacent
     * @param size
     *            - size of Independent set
     * @return "True", if the set is independent . "False" , if the set is not independent
     */
    // Check to see if the set if Independent
    public boolean FesibleSet(int[][] graph, ArrayList<Integer> Independentset, int size) {

        for (int i = 0; i < Independentset.size(); i++) {
            for (int j = 1; j < Independentset.size(); j++) {
                // if two vertices have an edge between we return false. Note* , 1 represents an edge in the matrix
                if (graph[Independentset.get(i)][Independentset.get(j)] == 1) {
                    return false;
                }
            }
        }
        // return true there are no edges between the vertices in the set.
        return true;

    }
}
```

Conclusion

In this paper, we dived deep into the history of Bitcoin and how it became a reality. The Bitcoin network tackles the double-spending problem by asking miners to assemble Bitcoin blocks for the blockchain in order to secure security. We have shown how we can translate the process of assembling a Bitcoin block into the Independent Set Decision Problem. To show the NP-completeness we can do it by proving the problem is NP and NP-hard. Using a third party python library, NetworkX, we are able to approximate a solution for the NP-hard version of the Independent Set Decision Problem. We also explored the Co-NP of the problem where the result will be "NO" when the answer is "YES" in NP. In order to find a solution, we touch on different well-known algorithms such as The Greedy Algorithm, Integer Linear Programming, and Bellman-Ford Algorithm. There is a special case when the graph is a tree and using the Greedy Algorithm can be solved in polynomial time. In addition, by converting Integer Linear Programming to Linear Programming we can also find a solution in Polynomial-time. Lastly, we came up with a brute-force algorithm takes in a graph in the form of matrix ones and zeros and we are able to come up with a "YES" or "NO" answer.

References

Bernard, Z. (2018, November 10). Everything you need to know about Bitcoin, its mysterious origins, and the many alleged identities of its creator. Retrieved from https://www.businessinsider.com/bitcoin-history-cryptocurrency-satoshi-nakamoto-2017-12#dorian-nakamoto-nick-szabo-and-craig-wright-arent-the-only-ones-who-been-pinned-as-the-inventor-of-bitcoin-11.

Bitcoin is NP-hard. (n.d.). Retrieved from https://tsa-heidi.github.io/Bitcoin/.

Bitcoin mining the hard way: the algorithms, protocols, and bytes. (n.d.). Retrieved from http://www.righto.com/2014/02/bitcoin-mining-hard-way-algorithms.html.

Bitcoin mining is NP-hard. (n.d.). Retrieved from https://freedom-to-tinker.com/2014/10/27/bitcoin-mining-is-np-hard/.

Branch Algorithm Kratsch, D. (n.d.). Retrieved from https://www-sop.inria.fr/mascotte/seminaires/AGAPE/lecture_notes/Agape 09 - Dieter Kratsch - B3e

Fortney, L. (2019, September 20). Blockchain Explained. Retrieved from https://www.investopedia.com/terms/b/blockchain.asp.

Independent set (graph theory). (2019, November 13). Retrieved from https://en.wikipedia.org/wiki/Independent_set_(graph_theory).

Installing¶. (n.d.). Retrieved from https://matplotlib.org/2.0.2/users/installing.html.

Independent set (graph theory). (2019, November 13). Retrieved from https://en.wikipedia.org/wiki/Independent_set_(graph_theory).

Kettley, S. (2017, December 21). Bitcoin price: How many bitcoin are there and when will the popular crypto token run out? Retrieved from https://www.express.co.uk/finance/city/895187/bitcoin-price-how-many-bitcoin-are-there-when-will-cryptocurrency-token-run-out.

Mathieu Mari, *Study of greedy algorithm for solving Maximum Independent Set problem*, (CNAM • ENS Rennes, 2017), 1-2, 4-6.

Maximum Independent Set. Retrieved from https://dspace.library.uu.nl/bitstream/handle/1874/853/c4.pdf

Mount, D. (2019). *CMSC 451: Lecture 20 NP-Completeness: 3SAT and Independent Set*. [ebook] CMSC 451. Available at: http://www.cs.umd.edu/class/fall2017/cmsc451-0101/Lects/lect20-np-3sat.pdf [Accessed 17 Oct. 2019].

Mostafa H. D.(2014, May 31). Maximum Independent Set Approximation Based on Bellman-Ford Algorithm

Semidoc.(2018, Jan 29). Greedy Algorithm for Maximum Independent Set. https://semidoc.github.io/greedyMIS

Studies Studio . (2018 , June 15). NP-completeness of Independent Set with Proof || By Studies Studio [Video file]. Retrieved from https://www.youtube.com/watch?v=ub8qsgPamqE

Jon. K. Algorithm Design. Retrieved from http://www.cs.princeton.edu/~wayne/kleinberg-tardos.

Yao Xin Feng	Yaoxin Feng
Keer Feng	[illegible]
Jonathan Hsiao	[illegible]
Samuel Lin	[illegible]
Sophia Shi	[illegible]