On NP-intermediate, Isomorphism problems, and Polynomial Hierarchy

Xin Lu

Abstract. Since being introduced, P and NP problems have been the main focus in complexity theory. Despite decades of efforts being put, it remains as an open problem whether $P = NP$. However, most professionals hold the belief that the two classes differ. In this paper, discussion on these two categories of problems will be hold, assuming $P \neq NP$. Work of Richard Ladner that shows the existence of NP-*intermediate* class of problems given $P \neq NP$ will first be discussed. Then, graph isomorphism problem, a commonly believed member of the intermediate class, and subgraph isomorphism problem will be discussed,primarily on a type of algorithm for them. Then, a brief discussion on polynomial hierarchy will be displayed, grounding on discussion of the isomorphism problems. At the end, a brief summary on significance and hardness of P $vs.NP$ problem will be put.

I. Background

Ever since computational models have been introduced, people have been working on classifying problems. The first generic question was whether all problems could be solved by a computational model. In 1936, Alan Turing showed that this question had a negative answer [1]. That is, there exists a problem that could not be solved by computational models. One of the example is the classic Halting problem. With this answer, people then turned toward problems that could be solved by computational models, or in other words, computable problems, and tried to analyze the hardness of these problems in terms of run time complexity. This is when class P and NP were introduced.

In 1970s, Stephen Cook published a paper, defining two classes of problems, P and NP, and proposed a question to the community: is $P = NP$ [2]? This problem, despite decades of efforts, remains to be an open problem. However, in order to understand these two classes, one will need to first understand some basic concepts in complexity theory.

Definition 1 (Language)**.** *A language L over Σ is simply a subset of Σ^*. Here, Σ can be considered as a set of finite alphabet. In other words, a language is just a collection of some strings formed by characters inside the given alphabet Σ.*

Each decidable problem can in fact be viewed as a language. For instance, consider the relative prime problem: given two inputs x and y, one wants to know whether x and y are relatively prime. One can view it as a language that collects all the strings $x\#y$ with x and y being integers that are relative prime. For example, a specific instance of this problem, like 2#3, will be in language L, as 2 and 3 are relatively prime.

For the rest of the paper, problem and language will be used interchangeably in order to ease the discussion.

Besides the definition of language, one will also want a formal definition for computational model in order to define P and NP. The formalization of computational models is provided by Turing in his thesis, which is called the Turing machine.

In informal term, one can consider a Turing machine as a machine composed by tapes, state register, head pointers and a finite table of instruction. The tape will be used to record input, output and work done so far and is in general considered to have infinite length. The state register records the current state the Turing machine is in. The head pointers point to the current symbol read on tapes and the finite table of instruction instructs the Turing machine based on current symbol pointed by the pointers and the state the Turing machine is in. Once the Turing machine reaches end state, it will be able to output the answer, which in general is either yes or not, identifying if the input is accepted by the Turing machine.

Formally, a Turing machine can be defined via a tuple of seven elements.

Definition 2 (Turing machine)**.** *A Turing machine M can be described by a tuple containing $Q, \Gamma, b, \Sigma, \delta, q_0, F$, with Q a finite, non-empty set of states, Γ a finite, non-empty set of tape alphabet symbols, $b \in \Gamma$ the special symbol reserved for blank space, Σ a finite, non-empty set of input symbols not containing b, q_0 the initial state and $F \subset Q$ a non-empty set of final states notifying the Turing machine to stop.*

δ, depending on whether it is a deterministic or non-deterministic Turing machine, will be function or a relation. It serves as the finite table of instruction that instructs the action of pointers, modification on content of tapes and update the state it is in.

With the definition of Turing machine, one can define a language in terms of Turing machine: $L(M) = \{x \mid x \text{ is accepted by } M\}$ is a language decided by M.

With these definitions, one can define the class P. In layman term, problems in class P can be considered as problems that could be solved by a computational model within polynomial time. The polynomial time indicates that the run time of the model can be written as a polynomial function of input length. An easy example will be to find the maximum number inside the input parsed in. This problem can be solved within $\mathcal{O}(n)$ times, as one can update maximum when enumerating over all

Definition 3 (class P)**.**

$$P = \{L(M) \mid M \textit{is a Turing machine that runs in polynomial time}\}$$

On the other hand, class NP is considered to be a collection of all problems that can easily verified, while not necessarily easy to solve. When one says it can be easily verified, it means that one can find a polynomial time Turing machine such that given input $x\#y$ where x is a description of the problem and y a proposal for this problem, the machine can check if y is a solution to problem x.

To formally define NP, one will not need the below definition. However, it is important for reducibility, which defines a special class of problem in NP. Thus, oracle Turing machine is introduced here with its definition shown as below.

Definition 4 (Oracle Turing machine)**.** *An oracle Turing machine is a Turing machine augmented by an oracle. During the execution of Turing machine, it can enter the state such that it will plug in certain input into the oracle and return the answer based on the output of the oracle.*

Naturally, one can view oracle as a black box. It empowers Turing machine the ability to process the input into certain format such that once it is solved by the oracle, the output of the Turing machine can be built.

Definition 5 (Class NP)**.** *A language L is in NP if there exists a polynomial time oracle Turing machine M such that x is accepted by M if and only if $x \in L$.*

It is obvious that if a problem is in P, then a problem is in NP. In other words, $P \subseteq NP$. However, whether this inclusion is a proper one is the famous $Pvs.NP$ problem. Yet, it will seem to be more natural to believe that not all computable problems can be solved in polynomial time. Hence, computer scientists in general hold the belief that $P \neq NP$, despite the lacking of actual proof.

With a more in-depth study in the class NP, Stephen Cook discovered a special class of problems within NP. That is, the NP*-complete* problems. These problems can be viewed as the upper bound of the level of difficulty for problems in NP. In other words, if a problem is in NP, then any solver that solves NP*-complete* problems will be able to solve it. Its rigorous definition will require knowledge on reducibility, and hence is postponed to the next section.

However, one can see that if any NP*-complete* problem is shown to be in P, then NP will collapse to P. Yet if NP turns out not to collapse to P, are these two classes close? In other words, if a problem does not belong to P, will it indicates that this problem is NP*-complete*? In 1975, Richard Ladner showed that under the assumption that $P \neq NP$, one can prove that there exists an intermediate class of problems between them. That is, there exist problems that belongs to neither P nor NP*-complete*, if $P \neq NP$.

The actual proof of this theorem is too long to be included in this paper. Yet, in the next section, one will discuss the idea behind this theorem, and at the same time offering a better preparation for later discussion.

II. Ladner's Theorem

With the belief that $P \neq NP$, one is likely to think about the distinction between these two classes. Are these two classes differ significantly in level of hardness, or are they actually close? In other words, is there a problem that falls between the two classes but lands in neither of them, or if a problem is in $NP \backslash P$, then it is NP*-complete*? This thought sparkles Richard Ladner to analyze the level of difficulty, later named as polynomial hierarchy, between P and NP. As a result, he showed that if $P \neq NP$, the two classes differed enough such that there would be an intermediate class such that problems in it were neither P nor NP*-complete*.

As laid out in previous section, to understand the idea behind Ladner's theorem, it is important to define *NP-complete* formally. As its definition depends on reducibility, one will first see the definition for reducibility.

Definition 6 (Reducibility)**.** *A problem A is said to be reducible to problem B if there is a polynomial time Turing machine T with oracle B such that on input x, T accepts x if and only if its output $T(x)$ being accepted by the oracle with respect to B.*

One can see that A being reducible to B can be viewed as saying A is an 'easier' problem compared to B, in the sense that solving B provides a solution to a twisted version of A. With this definition, one is able to define *NP-complete.*

Definition 7 (*NP-complete*)**.** *A problem A is said to be NP-complete if the following two conditions hold:*
i) $A \in NP$
ii) $\forall B \in NP$, B is reducible to A.

Before jumping into the idea behind Ladner's theorem, a few *NP-complete* problems are worth mentioning here. The first proven *NP-complete* problem is the so-called SAT problem, which asks for a satisfying assignment to a CNF formula. Since then, various *NP-complete* problems have been found by different computer scientists. In later section, this paper will focus on subgraph isomorphism problem, which is another proven *NP-complete* problem. Along with it, k-clique problem, a problem subgraph isomorphism problem can be reduced to (so it is also *NP-complete*), is also relatively important.

Now, one turns back to the main focus of this section: 'proof' of Ladner's theorem, whose statement is as below:

Theorem 1 (Ladner's Theorem [3])**.** *If $P \neq NP$, then there exists a problem in NP such that it is in neither P nor NP-complete.*

As mentioned earlier, this proof will be done through a proof by construction. In other words, using the assumption that $P \neq NP$, one will construct a problem that does not belong to either P or *NP-complete.*

That being said, one will like to construct a problem A such that $A \notin P$, A is reducible to B but B is not reducible to A, where B is another problem that falls in *NP-P*.

The first condition one wants to meet is probably that A is reducible to B. This condition can be easily met if one defines $A \subset B$ under certain criterion that can be tested within polynomial time. Then, a Turing machine that checks for this criterion will do the reduction job.

The hart part is to ensure that $A \notin P$ and B is not reducible to A. The key idea to formalize a construction meeting these two conditions is similar to the Cantor's diagonal argument. In order to say a problem A is not in P, one will like all Turing machines that decide languages in P fail to decide A. The way to do it follows strictly the diagonal argument: let $P_1, P_2, \cdots$ denotes an enumeration of all polynomial time Turing machine. If for each i, A contains a string $x \notin L_{P_i}$, then clearly A cannot be decided by P_i. If one is able to identify a systematic way to include a string x not in L_{P_i} in A for all i, then one will complete the proof for $A \notin P$.

Similar idea holds for showing B is not reducible to A. Saying that B is reducible to A means that there exists an oracle Turing machine $M(A)$ such that it decides B. Hence, to show that B is not reducible to A, one can again enumerates over all polynomial time Turing machine M_i with oracle A, denoting as $M_i(A)$. Then, for each $M_i(A)$, if A is constructed so that there exists a string $x \in B$ having its output $M_i(x) \notin A$, one will show that B is not decided by $M_i(A)$.

Conceptually, this is easy to see why in this way, the theorem can be proven. The point is, though, whether such A can be constructed, in a formal and systematical way. As mentioned, a formal proof of this theorem will be too long to include. The definition of the problem A provided by Ladner will be listed below, and a brief explanation on why it works will be supplied. Yet for interesting reader, they can turn towards the paper itself to see the full formal proof.

One will put $A = \{x \in B \mid |T(x)| \text{ is even}\}$. The key then becomes if T is a Turing machine that makes A a set satisfying conditions mentioned above. In the paper, T is defined as:

On input $x \neq 0^n$ of length $|x| = n$, output $T(0^n)$.
On input 0^n, if $n = 0$(so input is the empty string), output the empty string.
Otherwise, for n moves, reconstruct the sequence $T(\lambda), T(0), \cdots, T(0^m)$, with m being the last number computed within n moves.
If $|T(0^m)|$ is even, let i be the number such that $2i = |T(0^m)|$. Then for n steps, explore all strings z in lexicon order and see if $z \notin P_i$. If not, output 1^{2i}. Otherwise, output 1^{2i+1}.
If $|T(0^m)|$ is odd, let i be the number such that $2i + 1 = |T(0^m)|$. Perform the same thing as above, but this time see if $z \notin M_i(A)$. If not, output 1^{2i+1}, otherwise, output 1^{2i+2}.

It might be hard to understand in an intuitive way what T is doing. Yet, once you see one step, you will understand all others. Viewing the case $|T(0^m)|$ being even as marking T being in the state to construct A so that $A \neq P_i$, for the i defined in the step. In this case, it will consecutively output even length for any input, making these strings a member of A as long as it belongs to B. In other words, in this phrase, set A is made similar to B, where B is chosen to be in *NP-P*. As a result, there must be strings belong to B but not L_{P_i} and as A starts to look similar to B, the string z will eventually be found. Other steps could be explained similarly, and again detailed explanation is included in Ladner's paper.

Concluding this section, a detailed explanation on the intuition behind Ladner's proof is supplied. As a consequence of the theorem, one knows that if $P \neq NP$, there exist problems in the middle. However, if so, what are those problems? If one is able to identify these potential intermediate problems, analysis on their algorithms might be able to offer some insights into the entire complexity hierarchy.

One of the candidates is the graph isomorphism problem. It is one of the few problems that have not yet been shown to be in P or NP-*complete*, leading into a strong belief that it belongs to NP-*intermediate*.

On the other hand, subgraph isomorphism problem, a proven NP-*complete* problem, holds similarity with graph isomorphism problem: it is clear that a solver for subgraph isomorphism problem can easily solve graph isomorphism. Meanwhile, checking graph isomorphism seems to also be a necessary step in solving subgraph isomorphism problem. These two problems will, in fact, be the main focus of the following two sections.

In the next section, one will explore the current developed algorithm for subgraph isomorphism problem, capturing the key characteristic of the algorithm and study its complexity in terms of P and NP under certain condition.

III. Discussion on Subgraph Isomorphism Problem and Its Algorithms

Motivated by the similarity between subgraph isomorphism problem and graph isomorphism problem, one will like to compare these two problems in terms of their complexity, particularly since subgraph isomorphism problem is proven to be in NP-*complete*, while graph isomorphism's run time complexity remains as a mystery.

As planned above, in this section, discussion on the current algorithm for subgraph isomorphism will be displayed. In fact, one will characterizes the algorithm and defines it as a type. Then, a discussion on the lower bound of this algorithm will be held.

Before entering into the discussion, it is worthy to present a formal definition for the subgraph isomorphism problem.

Definition 8 (Subgraph Isomorphism Problem)**.** *Given input G and H, return whether or not H is a subgraph of G.*

Through years, there are various attempts on building up algorithms that will solve this problem.However, in general, these algorithms run in exponential time of the input size, unless the input is of a special type of graphs.

In this paper, the algorithm proposed by Cordella in 2004 will be the main focus. This algorithm starts of with an empty proposal (which refers to proposed solution). Then, relying on a checker function F, it will attempt to extend the proposal and F will check if the extended one remains a valid partial solution. In other words, if the extended proposal can not be extend to a full valid solution, F will propagate this branch. A detailed high-level description of the algorithm is as below [5] :

```
Input:  $G$, $H$, and initial node(state) $s$ with $M(s_0) = \emptyset$.
Output:  mappings between two graphs, null if DNE
$M(s)$:
   IF $M(s)$ covers all the nodes of $H$:
        OUTPUT $M(s)$
    ELSE:
        FOREACH $(n, m)$ edge can be extended
            IF $F(s, n, m)$ THEN
                Compute next node(state) $s^{'}$ obtained by this extension
                CALL $M(s^{'})$
    OUTPUT null
```

So far, most algorithms that attempt to solve subgraph isomorphism problems are of this manner. The question is, is this type of algorithm promising? In other words, will the optimal algorithm shares the same structure as Cordella's algorithm?

To think of that question, one should first consider the trivial algorithm, which constructs all possible proposals and then check for validity. This algorithm, in general, is considered not to be extendable to an algorithm with the optimal run time, as the number of subgraphs in G with k vertices are exponentially many. Hence, the question becomes whether by ensuring the current proposal is yet a valid partial solution, one is able to propagate enough number of proposals so that the algorithm runs in optimal run time.

To formalize this idea, this paper proposes the below definition to captures the characteristic of this type of algorithm.

Definition 9 (Propogating Algorithm). *An algorithm is said to be of propagating type if its structure fits in the below description:*

i) Starts at root node n_0 representing the initial state where the proposal is empty

ii) If at node n_i, propagation checker returns false for current proposal, propagate all proposals having the current proposal as a partial proposal.

iii) Otherise, extends the proposal by choosing available actions at node n_i, proceeds to the node directed by the action, repeat step ii).

Note, the propagation checker returns false for node n_i if and only if the proposal represented by node n_i cannot be extended to a full solution, grounding on the information contained in the constructed proposal.

It will be helpful to understand this algorithm if one considers the solution space (including both full and partial solutions) as a graph. Starting at initial node, the algorithm has no idea about the feature of a solution. Then, it will explore action in order. By action it refers to an atomic choice that extends the proposal. This will take one to the specific node representing the extended proposal depending on the action chosen. Hence, this process is like exploring the solution space as if the space is of a tree structure. To better illustrate the idea, consider the example displayed in Figure 1.

In this example, each action represents the inclusion of an edge (and corresponding vertices) into the subgraph constructed so far, and the next node induced by this action is the extended subgraph.

On the other hand, an example for the behavior of the propagation checker may be helpful. Hence, another example is provided in Figure 2.

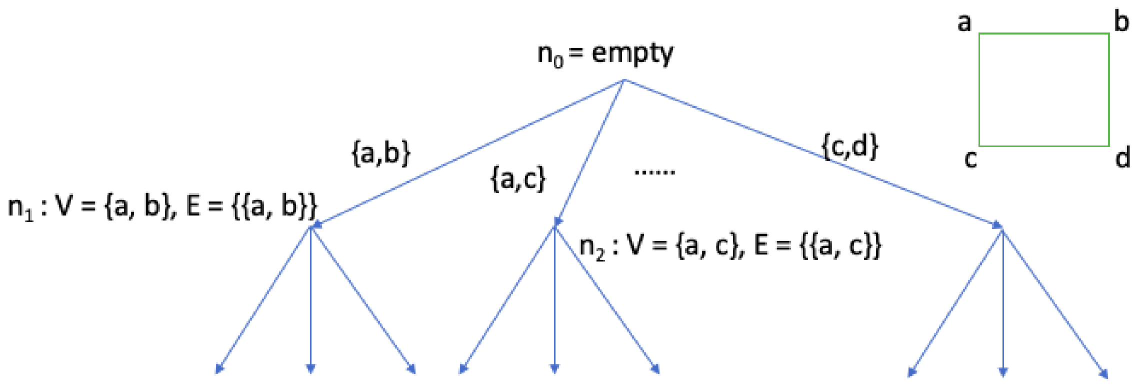

Figure 1: Solution Space for Square *abcd*

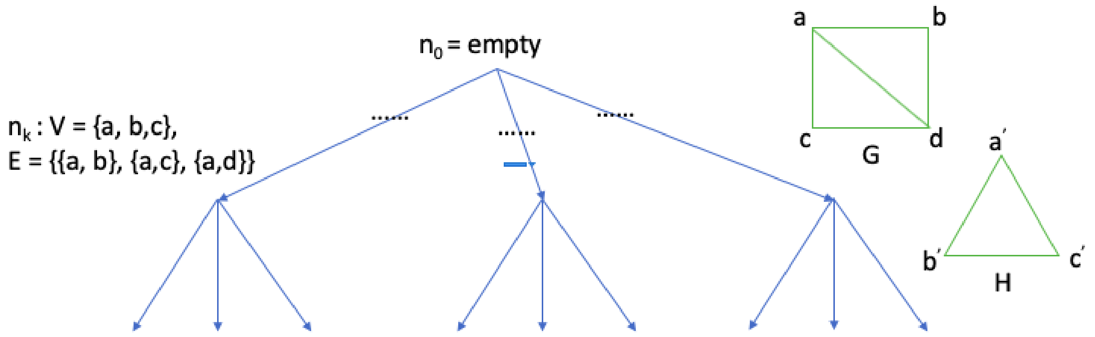

Figure 2: Propagate at n_k

Notice that when the algorithm reaches n_k, as by no way can a graph isomorphic to H has a node with degree > 2, the algorithm will propagate all nodes living in the subtree rooted at n_k.

Now, to analyze the potential of this type of algorithms, one can spot the two main parts that play a crucial role in it. The first is that how many nodes the algorithm has to explored. This determine the number of time the propagation checker will be invoked. The second is how expensive it is to run the checker. In the remaining part of this paper, these two portions will be decomposed and analyzed separately. The discussion will be laid out under the assumption that this type of algorithm can actually achieve the actual optimal run time for the problem, and hence directing the potential of this type of algorithm.

In this section, one will focus on the problem that how many time the propagation checker is called. Since it is hard to model the efficiency of the propagation checker, it will be hard to approach it directly. In fact, this paper does not have a solid answer to this question. Yet, instead, this paper consider the same question for the k-clique problem, which is an *NP-complete* problem. Hence, subgraph isomorphism problem and the k-clique problem should have the same solver, which indicates that the answer to this problem for k-clique might as well apply to subgraph isomorphism problem.

In order to proceed, of course, a definition for the k-clique problem is needed.

Definition 10 (k-clique). *A clique $C = (V', E')$ of a graph $G = (V, E)$ has $V' \subseteq V$ and $E' = \{\{v, w\} \mid v, w \in E\}$, where $E' \subseteq E$ as well. In other words, a clique is a complete subgraph of G.*

Naturally, one puts a k-clique of G as a k-complete subgraph of G.

With the definition of clique, one will be able to define k-clique problem.

Definition 11 (k-clique problem). *Given a graph G and an input k, finds if there exists a k-clique in graph G.*

Notice that both k-clique problem and subgraph isomorphism problem contains an exploration on subgraph of G, which is the primary intuition why these two problems are equivalent (and in fact they are). Hence, perhaps with further work, one will be able to nail down the bound for the number of time propagation checker is invoked for subgraph isomorphism problem as well.

Theorem 2. *If $P \neq NP$ and a propagating algorithm can be the optimal algorithm for k-clique problem, then enumerating all nodes visited by the propagating algorithm will be a problem in $NP - P$.*

Proof. Clearly, as k-clique problem is in NP, if the propagating algorithm achieves optimal run time, its subprocess should not exceed the total run time. Thus, enumerating all nodes visited by the propagating algorithm should be a problem in NP as well. Hence, if one can show that the enumeration can not be done in P, the proof is completed.

The proof will proceed by contradiction. Suppose the enumeration can be done in polynomial time (i.e. the problem is in P). Then, by definition of propagation algorithm, the propagation checker is called $f(n)$ times, where n is the input size and f a polynomial function.

As illustrated by the definition, a propagation checker return false if and only if the proposal representing by the current node cannot be extended into a full solution. As illustrated by the example, that is saying the current proposal is already inconsistent with the desire solution. For k-clique problem, one candidate proposal checker will be the one that checks if the current subgraph is partial isomorphic to k-complete graph. This is clear: if the graph is not partially isomorphic to the k-complete graph, then no matter how one extends the graph, the proposal contains a portion of graph that cannot be map isomorphically to the k-complete graph. Hence, any graph extended from this point will fail to be a k-clique. On the other hand, if the graph is partially isomorphic to the k-complete graph, simply appending the difference of the vertex and edge sets on current proposal and k-complete graph will yield a k-complete graph. Thus, this is a valid propagation checker.

Yet, this propagation checker only takes polynomial time. One only needs to verify if all nodes included in current partial solution has fewer than $\binom{k}{2}$ edges, which takes at most $\mathcal{O}(m+n)$ times where m marks the size of edge set of G. Hence, the optimal propagation algorithm should have the propagation checker with run time at most as bad as polynomial run time.

Then, if there are only polynomial amount of nodes being visited by the propagation algorithm of k-clique problem, it will run within polynomial time. Under $NP \neq P$ and the fact that *k-clique* problem is *NP-complete*, this cannot be the case. Hence, one has proven the statement. □

Thus, one has shown that for propagating algorithm, enumerating the nodes being visited by the algorithm will be a problem in *NP-P*. If one believe in the similarity among subgraph isomorphism problem and clique problem, one has the below conjecture.

Conjecture. *If the optimal algorithm for subgraph isomorphism problem can be a propagating algorithm, enumerating nodes visited by this propagating algorithm for subgraph ismorphism will be a problem in $NP - P$, supposing $P \neq NP$.*

However, only propagating when the algorithm finds an inconsistency seems to be not efficient enough. Consider graph G and H where almost all subgraphs of G are isomorphic to H except for one edge. If only propagating when inconsistency is met, for this example, the algorithm have to almost fully explored all possible subgraphs before encountering an inconsistency, assuming the worst case. This will indicate that such algorithm will remain to be of exponential run time, unless certain order is put on the exploration to avoid the worst case. In fact, Bonnici actually improves Cordella's algorithm by assigning a heuristic on the nodes such that exploration happens in a more favorable order [6]. Yet, despite this modification, the algorithm remains to have exponential run time.

All being said, it is hard to believe that the optimal algorithm for these problems could be done by such algorithm, if one believes that the optimal algorithm does not run in exponential time. This introduces another open problem that has not yet been proven: does $EXPTIME = NP$? Here, $EXPTIME$ captures all problems that can be decided in exponential time.

Concluding this section, the paper has shown that if the optimal algorithm can be situated into a propagating algorithm, the enumeration part, i.e., visiting all nodes not being propagated by the checker, is the hard part and will be a problem in $NP - P$, assuming inequality of the two classes. However, then, an example is given to show that by the defined checker, it seems unlikely that the propagating algorithm can be improved to reach below the exponential run time.

In the next section, the complexity of the checker will be analyzed. In particular, one will compare the propagation checker in propagating algorithm for subgraph isomorphism problem against the graph isomorphism. A few discussion will be put there, and then a theorem will be used to conclude the entire paper.

IV. Discussion on Graph Isomorphism and Polynomial Hierarchy

As mentioned in the last section, this section will focus on discussing the complexity of the propagation checker of subgraph isomorphism problem . This will be conducted via a comparison between the propagation checker and the graph isomorphism.

As usual, one should first see the formal definition of graph isomorphism.

Definition 12 (Graph Isomorphism)**.** *Given two graphs G and H, determine whether these two graphs are isomorphic.*

On the other hand, by the definition of propagation checker for algorithm solving subgraph isomorphism, one will naturally think of partial isomorphism. If a checker checks for existence of partial isomorphism between two graphs, it is easy to verify that it serves as a valid propagation checker for propagating algorithm.

However, a partial isomorphism checker is apparently stronger than graph isomorphism checker. In fact, one can easily reduce graph isomorphism problem to partial isomorphism problem.

Theorem 3. *Graph isomorphism is reducible to partial isomorphism problem.*

Proof. Given input G and H in the graph isomorphism problem. One can run a Turing machine that counts the number of vertices and edges in each graph. This can be done easily within polynomial time with respect to input size, as one only needs to visit all edges and nodes in both graphs.

If the number is distinct, then they cannot be isomorphic to each other, as isomorphism requires a bijective function while in this case the domain and range differs in size.

If the number is the same, then plug G and H into the partial isomorphism solver. If the partial isomorphism solver accepts, it means that H is isomorphic partially to graph G. Yet G and H have the same size, and thus induced an isomorphism. $\square$

Therefore, one can see that a propagating algorithm for subgraph isomorphism problem will have its propagation checker runs at a complexity at least as bad as solver for graph isomorphism. This seems to suggest that if graph isomorphism is NP-*complete*, then subgraph isomorphism will be harder than it non-trivially, if $P \neq NP$.

In fact, this intuition seems to be in right direction. It has been proven by Schoning that under the certain assumption, graph isomorphism will not be NP-*complete* [4]. To close this section, a brief introduction of this theorem will be provided.

In order to introduce this theorem, certain background on polynomial hierarchy will be necessary. Polynomial Hierarchy is a generalization on P, NP, and $coNP$ class of problems. In laymen term, it is a hierarchy of problems such that each layer of problems are constructed using previous layer and is anticipated to be of different classes, unless the hierarchy collapses. In fact, if $NP = P$, then polynomial hierarchy collapses completely.

There are various types of definitions for polynomial hierarchy. In here, the paper offers the definition of polynomial hierarchy used by Schoning Uwe.

Definition 13 (Polynomial Hierarchy). *Define* $\Delta_0^P = \Sigma_0^P = \prod_0^P = P$*, then one define, recursively,*

$$\Delta_{i+1}^P = P^{\Sigma_i^P} \quad \Sigma_{i+1}^P = NP^{\Sigma_i^P} \text{ and } \prod_{i+1}^P = coNP^{\Pi_i^P}$$

where P^A *is the set of decision problems solvable in polynomial time by a Turing machine augmented by an oracle for some complete problem in set A, and others are defined similarly.*

From the definition, a notice that should be mentioned here is that if indeed the propagating algorithm is an optimal algorithm for subgraph isomorphism, then with the assumption that graph isomorphism is NP-*complete*, one will ends up showing that subgraph isomorphism problem is actually in Σ_2^P if the enumeration part is in NP-P, as stated by the conjecture in early section. This is because the propagating algorithm can be decomposed as an oracle Turing machine T that runs the exploration part of the propagating algorithm and an oracle on the partial isomorphism problem, which is at least NP-*complete*. Hence, it falls in the definition of Σ_2^P. On the other hand, if the enumeration part can be shown to be NP-*complete*, this will indicate that in fact, Σ_2^P collapses to $\Sigma_1^P = NP$.

Yet, this conjecture is grounded on several unanswered questions and therefore will not be fully extended in this section. Rather, the theorem proven by Schoning on the complexity of graph isomorphism will be displayed [4].

Theorem 4. *Graph isomorphism cannot be* NP*-complete unless the polynomial hierarchy collapses to* Σ_2^P*.*

The proof of this theorem requires solid understanding on a few different classes of problems, such as $coNP$ and $coAM$. Hence, its proof will be out of the scoop of this paper and again is left to readers who are interested in it.

V. Conclusion

In this paper, one starts off with basic facts in complexity theory. Then, an in-depth look on the Ladner's theorem, which shows the existence of intermediate class given $P \neq NP$ is supplied. After this, a detailed exploration on current algorithm that solves subgraph isomorphism problem is conducted, with various analysis on it, suggesting it might not be close to the optimal run time for this problem unless $NP = EXPTIME$. Then, a brief discussion on polynomial hierarchy and a theorem about complexity of the graph isomorphism problem is mentioned.

In general, there are a lot of extensions can be made grounded on the assumption that $P \neq NP$. However, the direction to prove this assumption itself remains unclear. Initially, this paper starts off to investigate whether the complexity of graph isomorphism problem might offer some hints on differing P and NP, while in the end the statement that $P \neq NP$ is at least as strong as saying graph isomorphism is NP-*intermediate*. The difficulty here is that there exists not yet a statement that seems to be stronger than the statement $P \neq NP$, or equivalent but with a clearer direction to approach.

Despite the little progress on this problem, the introduction of quantum computing might relieve people from this long lasting open problem. If a quantum computer is devised, the run time difference between P and NP will in fact be as of less important, as it bestows a more powerful computing machine. Either way, the unexplored field in complexity theory remains wide open and deserves successive input.

References

[1] Turing Alan Mathison, "On computable numbers, with an application to the Entscheidungsproblem.", J. of Math 58.345-363 (1936): 5.

[2] Cook Stephen, "The P versus NP problem.", The millennium prize problems (2006): 87-104.

[3] Ladner Richard E., "On the structure of polynomial time reducibility.", Journal of the ACM (JACM) 22.1 (1975): 155-171.

[4] Schöning Uwe, "Graph isomorphism is in the low hierarchy.", Journal of Computer and System Sciences 37.3 (1988): 312-323.

[5] Cordella Luigi Pietro et al, "An improved algorithm for matching large graphs.", 3rd IAPR-TC15 workshop on graph-based representations in pattern recognition(2001).

[6] Bonnici Vincenzo et al, "A subgraph isomorphism algorithm and its application to biochemical data.", BMC bioinformatics 14.S7 (2013): S13.