On NP-intermediate, Isomorphism problems, and Polynomial Hierarchy

Xin Lu

Abstract. Since being introduced, *P* and *NP*

De nition 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.

De nition 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 2 L.*

It is obvious that if a problem is in P, then a problem is in NP. In other words, P 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 di culty 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 de nition 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 o ering a better preparation for later discussion.

II. Ladner's Theorem

With the belief that $P \neq NP$, one is likely 104(P)c/F1si513(b)-27(et)26gTf 20.811 /F1Tf 45.6uss275(wil97

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 $T(\epsilon); T(0); \ldots; T(0^m)$, 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 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 de nes 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 de nition for the subgraph isomorphism problem.

De nition 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 $M(s_0) = ;$.
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 rst 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 de nition to captures the characteristic of this type of algorithm.

De nition 9 (Propogating Algorithm). *An algorithm is said to be of propagating type if its structure ts 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.

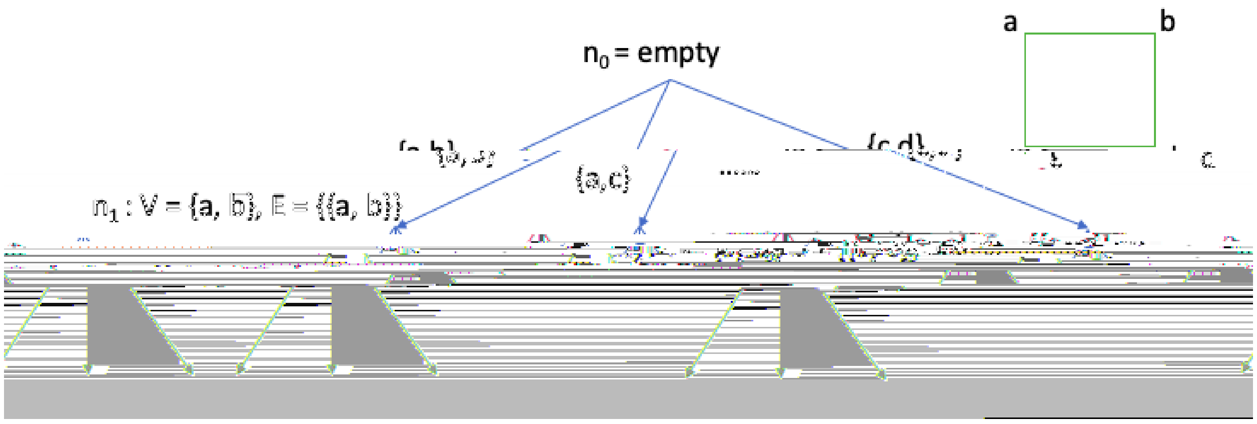

Figure 1: Solution Space for Square *abcd*

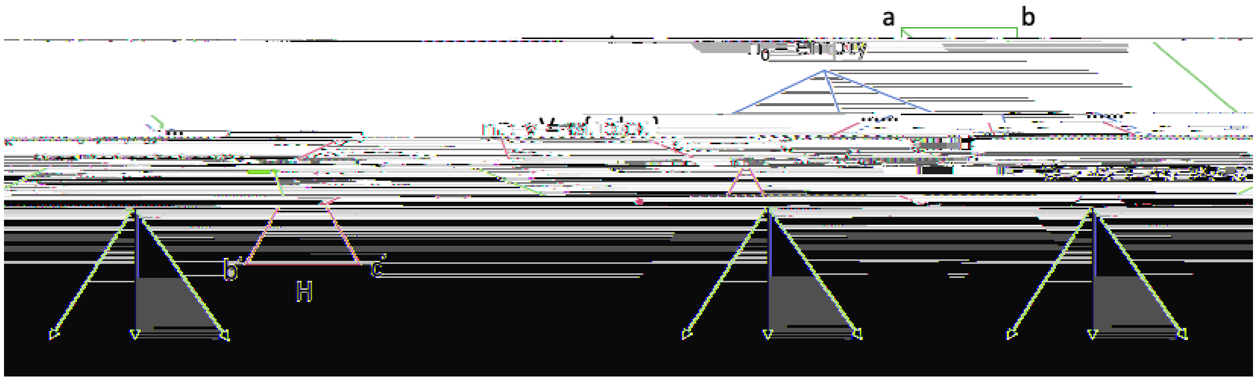

Figure 2: Propagate at n_k

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 $NP \setminus P$.*

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 de nition, 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 di erence 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 $\substack{k \\ 2}$ edges, which takes at most $O(m + n)$ 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 $NP \quad P$, supposing $P \neq NP$.*

However, only propagating when the algorithm nds an inconsistency seems to be not e cient 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 modi cation, 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 sit-

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 backo-

V. Conclusion