2011

The INDEPENDENT SET Decision Problem is NP-complete

Andrew Bristow IV
Virginia Commonwealth University

College of Humanities and Sciences
Virginia Commonwealth University

This is to certify that the thesis prepared by Andrew Schuyler Bristow IV titled "The INDEPENDENT SET Decision Problem is NP-complete" has been approved by his or her committee as satisfactory completion of the thesis requirement for the degree of Master of Science.

Craig Larson, College of Humanities and Sciences

Aimee Ellington, College of Humanities and Sciences

J. Paul Brooks, College of Humanities and Sciences

John F. Berglund, Graduate Chair, Mathematics and Applied Mathematics

---

James S. Coleman, Dean, College of Humanities and Sciences

F. Douglas Boudinot, Graduate Dean

Date

The INDEPENDENT SET Decision Problem is NP-complete

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at Virginia Commonwealth University.

by

Andrew Schuyler Bristow IV
Master of Science

Director: Craig Larson, Assistant Professor
Department of Mathematics and Applied Mathematics

Virginia Commonwealth University
Richmond, Virginia
August 2011

Acknowledgment

I would like to express my deepest thanks to my thesis advisor Dr. Craig Larson, for the countless hours he has spent working with me on this thesis. His support and humor throughout this process has been greatly appreciated. I would also like to thank my committee members, Dr. Aimee Ellington and Dr. J. Paul Brooks, for their assistance and support. I sincerely appreciate their effort in helping me accomplish this endeavor.

Lastly, I would be remiss if I forgot to thank my biggest supporter in my effort to earn this degree. To my wife, Amy, thank you for all your encouragment in allowing me to finish this journey I started some many years ago. It is with your support that this achievement is possible.

Contents

Abstract

THE INDEPENDENT SET DECISION PROBLEM IS NP-COMPLETE

By Andrew Schuyler Bristow IV, Master of Science.

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at Virginia Commonwealth University.

Virginia Commonwealth University, 2011.

Director: Craig Larson, Assistant Professor, Department of Mathematics and Applied Mathematics.

In the 1970's computer scientists developed the theory of computational complexity. Some problems seemed hard-to-compute, while others were easy. It turned out that many of the hard problems were equally hard in a way that could be precisely specified. They became known as the NP-complete problems. The SATISFIABILITY problem (SAT) was the first problem to be proved NP-complete in 1971. Since then numerous other hard-to-solve problems have been proved to be in NP-complete. In this paper we will examine the problem of how to find a maximum independent set of vertices for a graph. This problem is known as Maximum Independent Set (MIS) for a graph. The corresponding decision problem for MIS is the question, given an integer K, is there a independent set with at least K vertices? This decision problem is INDEPENDENT SET (IS). The intention of this paper is to show through polynomial transformation that IS is in the set of NP-complete Problems. We intend to show that 3SAT is NP-complete and then using this fact, that IS is NP-complete.

Introduction

This chapter is designed to inform the reader on the basic terminology and definitions associated with the mathematical field of graph theory. The author assumes that the reader has little to no background knowledge in this area. My attempt here is to build a basic foundation in graph theory knowledge that will be both helpful and necessary for the later examination of certain graph theory decision problems that are NP-complete.

1.1 Graph Basics

A **graph** is made up of a set of points and line segments or curves connecting certain pairs of vertices. Each point in a graph is called a **vertex**. Each line segment or curve that connects two vertices is called an **edge**. An example of a graph appears in Figure **??** on the following page. At this point, it is necessary to provide a more technical definition of a graph.

DEFINITION 1.1. A graph ***G*** = (V,E) where ***V*** represents the set of vertices and ***E*** represents the set of edges in the graph which are two-element subsets of V. If $V = \{v_1, ..., v_n\}$ and $\{v_i, v_j\} \in E$, we will write edges as (v_i, v_j).

For the graph in Figure **??** ,
$V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$ and $E = \{(v_4, v_6), (v_4, v_5), (v_4, v_3), (v_3, v_2), (v_5, v_2), (v_5, v_1), (v_2, v_1)\}$.

In a graph, when two vertices are connected by an edge the vertices are known as **adjacent vertices**. In Figure **??**, you can see that vertex 4 is adjacent with vertices 6, 3 and 5. Vertices 1, 2, 3, 4, 5 and 6 are all adjacent to another vertex. It is only vertex 7 that has no adjacent vertices.

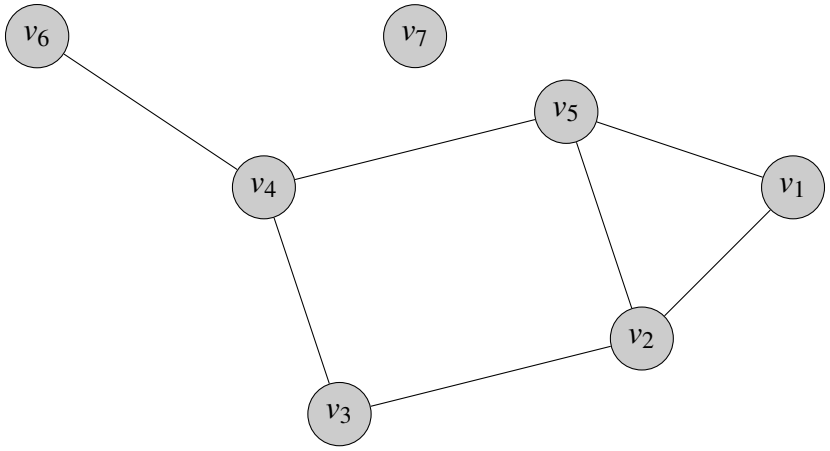

Figure 1.1: Example of a graph.

DEFINITION 1.2. The **degree of a vertex**, *deg(v)*, is the number of edges incident to that vertex plus twice the number of loops.

Looking at Figure **??**, one can determine the degrees of all seven vertices. You should be able to see that $deg(v_1) = 2$, $deg(v_2) = 3$, $deg(v_3) = 2$, $deg(v_4) = 3$, $deg(v_5) = 3$, $deg(v_6) = 1$ and $deg(v_7) = 0$. One important thing about the degrees of a graph is that the sum of all of the degrees is equal to twice the number of edges.

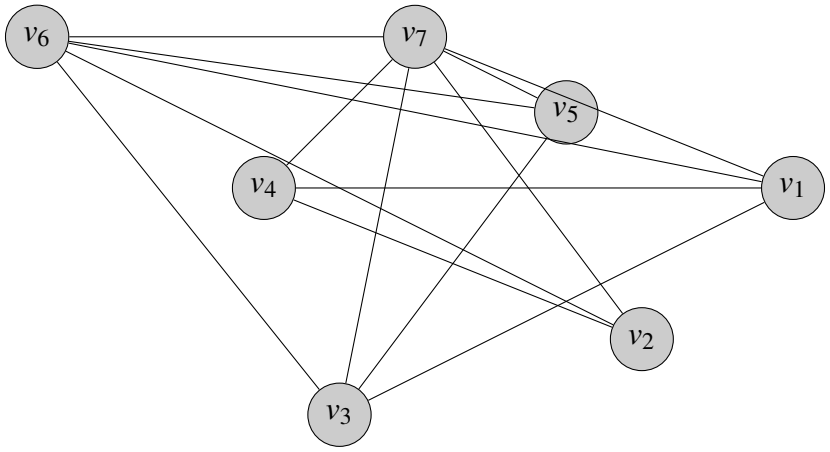

Figure 1.2: Complement of graph in figure **??**

The **complement** of a graph G, is another graph G^c, which has the same set of vertices as G, but includes only those edges that are not in G. More formally, for every pair of vertices $v, w \in V$, $(v,w) \in E(G^c)$ if and only if $(v,w) \notin E(G)$. In Figure **??** you will find the complement of the graph presented in Figure **??** . Notice how in the complement each vertex now shares an edge with all of it's non-adjacent vertices in the original graph. Also, adjacent vertices from the original graph become non-adjacent in the compliment.

Now that some basic terminology has been introduced I will briefly present some examples of common graph classes. Each graph class will be presented by listing the definition of the graph followed by an visual example. Formulas for the independence number of these graph classes will be presented in the following section.

DEFINITION 1.3. A **complete graph**, K_n, is a graph in which every two distinct vertices are adjacent.

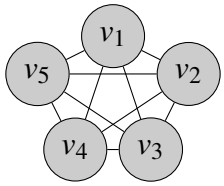

Figure 1.3: **Complete Graph** K_5.

A complete graph with 5 vertices, K_5, is given in **Figure ??** below. An important aspect to keep in mind for later is that each vertex shares an edge with all of the other vertices in the graph. This fact about any K_n will be helpful in computing the independence number in the next section.

DEFINITION 1.4. A **path**, P_n, is a simple connected graph with n vertices, $\{v_1, v_2, ..., v_n\}$, and $n-1$ edges, $\{(v_1,v_2),(v_2,v_3),...,(v_{n-1},v_n)\}$.

Intuitively, think of a path as a graph that can be drawn so that all of the vertices and

edges lie on a single straight line. Path graphs P_4 and P_5 can be seen in the figure below. It should be known that the placement of the two graphs in figure **??** was done on purpose. In the next section we will examine the independence number for this graph and the formula will actually depend upon whether the number of vertices in the graph is even or odd. By adding one additional edge to a path we can create a new type of graph called a cycle, which is introduced next.

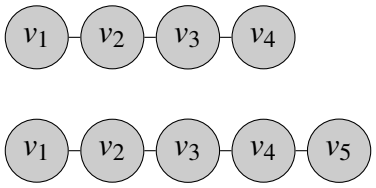

Figure 1.4: **Path graphs** P_4 and P_5

DEFINITION 1.5. A **cycle graph** C_n is a simple connected graph with n vertices and $n-1$ edges where $V(C_n) = \{v_1, v_2, v_3, ..., v_n\}$ and $E(C_n) = \{(v_1, v_2), (v_2, v_3), ...(v_n - 1, v_n), (v_n, v_1)\}$.

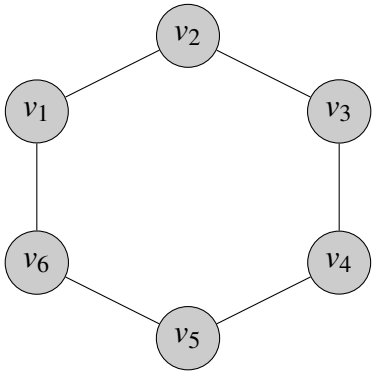

Figure 1.5: **Cycle Graph** C_6.

DEFINITION 1.6. A **star** S_n is the complete bipartite graph $K_{1,k}$ which is a tree with one internal node and k leaves.

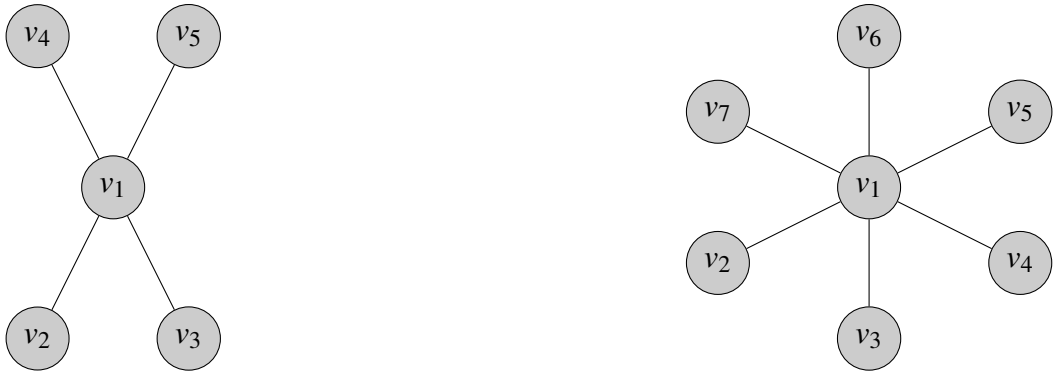

Figure 1.6: **Star graphs** S_5 and S_7.

The star graph is made up of n vertices where one central vertex is adjacent to all the remaining vertices in the graph. One thing of importance to note that will help us later is the fact that all of the non-central vertices are non adjacent. This will make the independence number for any star graph easy to compute.

DEFINITION 1.7. A **complete bipartite graph** $K_{m,n}$ is the simple graph on $m+n$ vertices where $V(K_{m,n}) = \{u_1, u_2, ..., u_m\} \cup \{v_1, v_2, ...v_n\}, E(K_{m,n}) = \big\{\big\{u_i, v_j\big\} : 1 \leq i \leq m, 1 \leq j \leq n\big\}$.

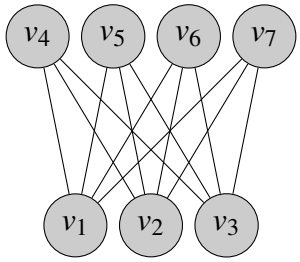

Figure 1.7: **Complete Bipartite Graph** $K_{3,4}$

1.2 Independence Number

To be able to determine the independence number of a graph, you must first be able to identify an independent vertex set in a graph. An **independent vertex set** in a graph is a set

of vertices that does not include any pair of adjacent vertices.

DEFINITION 1.8. The **independence number of a graph**, α , is the number of vertices in a maximum independent set.

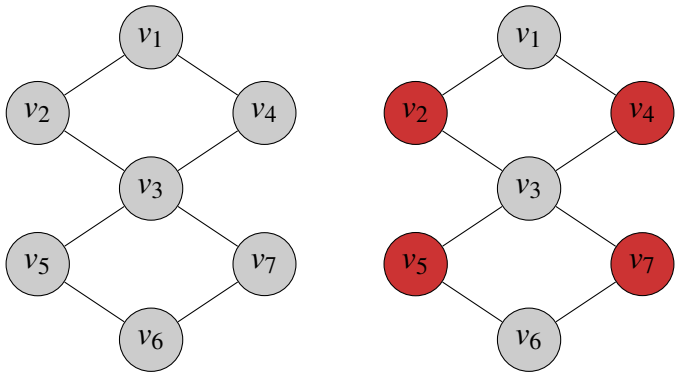

Figure 1.8: Example of a graph and it's maximum independent set of vertices.

In Figure **??** you see the maximum independent set of vertices shaded in red. For this particular graph there are many sets of independent vertices. For example the sets $\{v_1, v_5, v_7\}$ and $\{v_2, v_4\}$ and $\{v_1, v_3, v_6\}$ are all independent sets of vertices. However, for that graph, only the set containing vertices 2, 4, 5, and 7 is maximum. A **maximum independent set** of vertices is the largest possible set of independent vertices for a given graph, G. Therefore the graph in Figure **??** has an independence number equal to 4.

To solidify our understanding of calculating the independence number of a graph let's look back at some of the common graph classes introduced in Section 1.1. We begin by looking back at the complete graph K_n and the example of K_5 given in Figure **??**. From the definition of a complete graph we know that all vertices in a complete graph are adjacent. Thus if all of the vertices are adjacent, a maximum independent set can include just one of the vertices so we can conclude that for any complete graph K_n , $\alpha = 1$.

Next, we will examine the path graphs P_4 and P_5 presented in Figure **??**. It should be easy to see that the $\alpha = 2$ for P_4 and that $\alpha = 3$ for P_5. An easy way to calculate α for any

path is to count the number of odd vertices in the graph. Note this does not be count all the vertices of odd degree but instead to count the vertices in positions 1, 3, 5, ... until you reach the end vertex of the path. This process can be simplified into the following formula. For any path P_n where n is the number of vertices, $\alpha(P_n) \leq \frac{n}{2}$ if n is even and $\alpha(P_n) \leq \frac{n+1}{2}$ if n is odd. It is clear that there is no larger independent set. Thus $\alpha(P_n) = \frac{n}{2}$ if n is even and $\alpha(P_n) = \frac{n+1}{2}$ if n is odd.

Notice that the graph C_n differs from the graph P_n in the simple fact that v_1 and v_n are connected. Therefore, one might conclude that the independence numbers for these two graphs might in fact turn out to be the same or very close to the same. In fact, that is half-way true. To calculate α for any C_n we need to look at the number of vertices in the graph just as we did for the path. If the number of vertices in the graph is even, we can calculate α by taking number of vertices and dividing by 2. If the number of vertices in the graph is odd, $\alpha = \frac{n-1}{2}$.

A star graph, S_n, is another easy graph to calculate the independence number for. All you have to do is take the number of vertices and subtract 1. So for any S_n graph $\alpha(S_n) = n - 1$.

The last graph introduced in the first section was $K_{m,n}$, the complete bipartite graph. Since any complete bipartite graph by definition has it's set of vertices split into two sets of non-adjacent vertices of size m and n, the maximum independent vertex set for the graph is either going to have to be the set $\{u_1, u_2, ..., u_m\}$ or $\{v_1, v_2, ..., v_n\}$ where $u, v \in V$. So $\alpha(K_{m,n}) = max\{m, n\}$.

1.3 Independence Number Relations In A Graph

The independence number of a graph is related to many other basic graph invariants including the domination number, the chromatic number, the clique number and the matching number. In this section we explore these relationships. We begin with the domination number of a

graph.

DEFINITION 1.9. A **dominating set** of a graph is the set of vertices in graph (G) such that every vertex in V(G) is either in D or adjacent to a vertex in D.The **domination number of a graph**, γ(G), is the cardinality of a minimum dominating set of G.

THEOREM 1.10. For any graph, $\gamma(G) \leq \alpha(G)$.

Proof. Let G be a graph. Let V be the set of all vertices in G. Let S be a maximum set of independent vertices. Then $|S| = \alpha$, where α is the independence number of a graph. Since S is maximum, any vertex in $G - S$ is a neighbor of some vertex in S. Therefore S is also a dominating set. So either $\gamma \leq |S|$. So $\gamma \leq \alpha$. □

An important concept in graph theory is that of graph coloring. A proper coloring of a graph is a coloring in which any two vertices joined by an edge are assigned two different colors. A graph is vertex k-colorable if it has a proper vertex k-coloring. We are concerned with the smallest number of colors that can be used to color a graph so that no adjacent vertices share the same color. This concept leads us to our next definition and proof relating to the chromatic number of a graph.

DEFINITION 1.11. The **chromatic number of a graph**, χ(G), is the minimum number of different colors required for a proper vertex-coloring of G.

THEOREM 1.12. For any graph, $\chi \cdot \alpha \geq n$.

Proof. If χ(G) = m, then the set of vertices, V, of graph G, can be partioned into m color classes, $V_1, V_2, ..., V_m$ each of which is an independent set of vertices. Therefore,

$$|V_1| + |V_2| + ... + |V_m| = n. \tag{1.1}$$

Let $c = \max\{|V_1|, |V_2|, ..., |V_m|\}$. Note that $\alpha \geq |V_i|$ for every $i \in \{1, 2, ..., n\}$, so, $\alpha \geq c$. So $\alpha \cdot m \geq c \cdot m \geq |V_1| + |V_2| + ... + |V_m| = n$, thus proving the result. □

Another concept that will be important to us is finding out what the clique number of a graph is.

DEFINITION 1.13. A **clique** is a set of pairwise adjacent vertices. The **clique number of a graph**, $\omega(G)$, is the number of vertices in a maximum clique in G.

THEOREM 1.14. $\alpha(G) = \omega(G^c)$.

Proof. Let G be a graph. Let V = set of all vertices in G. Then α by definition is equal to the size of the largest set of independent vertices in G. Since G^c consists of all of the vertices in G combined with all of the non-edges in G, the largest independent set in G becomes a clique in G^c. A clique, by definition, is a set of all adjacent, connected, vertices in the graph. Thus $\omega(G^c) \geq \alpha(G)$. Similarly, the largest clique in G^c becomes an independent set in G. Thus $\alpha(G) \geq \omega(G^c)$. Thus $\alpha(G) = \omega(G^c)$. □

The last relationship we will look at involves the matching number of a graph. A **matching** in $G = (V, E)$ is a set $M \subseteq E$ of pairwise non-adjacent edges. The **matching number of a graph**, $\mu(G)$, is the size of any largest matching in G.

THEOREM 1.15. For any graph G,

$$n - 2\mu \leq \alpha \leq n - \mu \tag{1.2}$$

Proof. We begin by showing $\alpha \leq n - \mu$. Let $G = (V, E)$ where M is a maximum matching in G. So, $\mu = |M|$. At most one vertex from each edge in M is in any independent set. So, $\alpha \leq |V \setminus V(M)| + |M| = (n - 2\mu) + \mu = n - \mu$.

Now we show $n - 2\mu \leq \alpha$. Note that $V \setminus V(M)$ is an independent set. So $\alpha \geq |V \setminus V(M)| = n - 2\mu$. □

1.4 Computing Independence Number

Computing the independence number for a graph can be very time-consuming. Since the 1970's mathematicians and computer scientists have worked to shorten the length of time it takes to calculate α for a given graph. The least effective and most time consuming algorithm used to compute the independence number of a graph is called the **naive algorithm**. For any $G(V,E)$ with $|V| = n$, there are exactly 2^n subsets of V. The naive algorithm will go through and perform two tasks for each subset. The first task is to check that the subset of vertices is independent. If the vertices are independent, it then will calculate and store the independence number for that subset. The naive algorithm continues this process until it has examined all of the 2^n subsets of V. Once completed, it takes the largest independence number from the set of subsets and sets that equal to the independence number of the graph. This process has a worst-case time bound $O(2^n)$. For now it will suffice to just examine the function inside of the parentheses of the time bound function, O and see that it is exponential and not polynomial. Repeated attempts to decrease the amount of time it took to compute the independence number of a graph continued until a breakthrough came in the late 1970's.

In 1976, Tarjan and Trojanowski [**?**] found an algorithm that could find a maximum independent set for a graph with n-vertices in $O(2^{n/3})$ time. In their algorithm, they start with a graph $G(V,E)$. They then select some $v \in V$ and create a set of vertices that are adjacent to vertex v, $A(v)$. They then reason that any maximum independent set for G either contains v or does not. Therefore, any maximum independent set of G is either v combined with a maximum independent set in $G(V - \{v\} - A(v))$ or it is a maximum independent set in $G(V - v)$. So $\alpha(G) = 1 + \alpha(G - \{v\} - A(v))$ or $\alpha(G) = \alpha(G - \{v\})$. They then extended this idea to instead of just looking at single vertices, to examine a subset, S, of vertices from V. Then from S create a set of vertices adjacent to those vertices in S called $A(S)$. If $S \subseteq V$, then any maximum independent set I in G consists of an independent set

$I \cap S$ in $G(S)$ and a maximum independent set $I - S$ in $G(V - S - A(I))$. The algorithm then finds each independent set J in S and a paired maximum independent set in $G(V - S - A(J))$. The last thing that speeds up the calculation is that the algorithm also uses a technique which compares independent subsets in S and eliminates independent subsets within S that have smaller independence numbers. Clearly, this algorithm is faster than the naive algorithm since $2^{n/3} \leq 2^n$ but it might help to see an example of how this algorithm works.

Let $G = (V, E)$ be the graph from Figure **??** and let $S = \{v_2, v_3, v_4, v_6, v_7\}$. Then by their algorithm $A(v_2) = \{v_1, v_3, v_5\}$ and $A(S) = \cup_{v \in S} A(v)$. So $A(S) = \{v_1, v_2, v_3, v_4, v_5, v_6\}$. Now let $I = \{v_2, v_4, v_7\}$ and $J = \{v_3, v_6, v_7\}$. Clearly both I and J are independent sets in $G(S)$. Let I' be an independent set in $V - S$ such that $I \cup I'$ is independent. We define J' similarly, $J' \subseteq V - S$ such that $J \cup J'$ is independent. For our example, $V - S = \{v_1, v_5\}$, so $I' = \{\oslash\}$ and $J' = \{v_5\}$. Examining the sizes of the joined sets $I \cup I'$ and $J \cup J'$ we see that $|J \cup J'| \geq |I \cup I'|$. Here $4 \geq 3$. This fact allows us to eliminate the process of using this algorithm on all independent subsets of I since we get an independent set at least as large by running the algorithm for J. Thus eliminating the need to check every subset like in the naive algorithm. It should be noted that we could have used $J' = \{v_1\}$ instead but the outcome would have been the same.

The latest time reduction improvements came in 1986 when Robson made a observation that reduced calculation time to $O(2^{0.276n})$ [**?**]. Robson's algorithm is similar to the one presented by Tarjan and Trojanowski [**?**] but uses detailed observations based on the possible subgraphs around a chosen vertex, connected graphs, and exponential space. To date, Robson's algorithm is the fastest known algorithm to calculate the size of a maximum independent set of vertices from an n vertex graph.

Decision Problems and Computation

2.1 Decision Problems

In order to understand what it means for a decision problem to be in either class P, NP or NP complete, we need to start by examining what a decision problem actually is. A decision problem, Π, is a problem with only two possible solutions, either a yes or no for a given instance. Each decision problem is broken into two main parts. The first part is called the "instance" which basically specifies the components that make up the problem. The second part called the "question" asks a yes or no question based upon the information from the given instance. Below I will give the standard form for such a problem followed by a very specific example for clarification.

INDEPENDENT SET Π_{IS}
INSTANCE: Given a specific graph $G(V,E)$ and a specific positive integer $J \leq |V|$.
QUESTION: Is there a independent set $V' \subseteq V$ so that $|V'| \geq J$.

For our example let's begin by looking at the graph first presented in Figure **??**. By using the graph in Figure **??** we can create an independence number decision problem and it could look something like this:

INDEPENDENT SET Π_{IS}
INSTANCE: Given graph $G(V,E)$ in Figure **??** and the integer 3.

QUESTION: Is there an independent set $V' \subseteq V(G)$ so that $|V'| \geq 3$?

In this particular problem we are asking the question " Is there an independent set V' so that $|V'| \geq 3$?". It should be quite obvious to see that based on the given information in the instance and the question asked we are positioned to only recieve a yes or no answer. In this particular instance let $V' = \{v_1, v_3, v_6, v_7\}$ and the answer is yes, $|V'| = 4 \geq 3$. Can the independence number $\alpha(G)$ of a graph G be computed efficiently? It is enough to determine if the decision problem IS can be computed efficiently. Given a graph G, if we can answer the decision problems for $k = 1, k = 2, ..., k = n$ then the largest k where the answer is yes equals $\alpha(G)$. For each decision problem there exists an infinite set of instances with notation D_Π. A subset of D_Π is the set of yes instances with notation Y_Π. It should be noted that our instance from the preceding example belongs to the set Y_{IS}.

Each decision problem has associated with it an encoding scheme and a language. In order for a decision problem like the one given in the previous example to be evaluated by a computional device we need to change the problem into something that would be recognizable by a computer. The way that we do this is that we use an **encoding scheme**, e, which is a way to transform a decision problem into a computer input in such a way that we can represent any possible instance of a decision problem by a finite string of symbols. We will look at a few examples of how we can encode the graph in Figure **??** into something that might be recognizable to a computer.

The first encoding scheme that we can look at is the encoding scheme in which a graph is changed into a string of vertices and edges. So taking the graph in Figure **??** we can change it into something that looks like:

$V[1]V[2]V[3]V[4]V[5]V[6]V[7](V[1]V[2])(V[1]V[5])(V[2]V[5])(V[4]V[5])(V[2]V[3])(V[3]V[4])(V[4]V[6])$

In this encoding scheme all of the vertices of the graph are entered first. For example, vertex $v_1 = V[1]$. After all of the vertices have been entered then edges are entered in sets of parentheses where two vertices sharing an edge are entered in a set of parenthses. This is

continued until all edges have been listed. This encoding scheme is refered to as the *Vertex list, Edge list* encoding scheme by Garey and Johnson.

Another encoding scheme offered by Garey and Johnson is the encoding scheme *Neighbor list*. In the neighbor list each set of parentheses represents a single vertex and the vertices listed inside each set of parentheses are it's adjacent vertices. For example the graph in Figure **??** would look like the following if it were to be encoded using the neighbor list:
$(V[2]V[5])(V[1]V[3]V[5])(V[2]V[4])(V[3]V[5]V[6])(V[1]V[2]V[4])(V[4])()$
In this encoding scheme v_1 is represented by the first set of parentheses. Inside that set of parentheses are V[2] and V[5]. That means that v_1 shares an edge with both v_2 and v_5. Note that the parentheses representing v_7 is empty. In the orginal graph v_7 was non-adjacent to all other vertices therefore it would not have any vertices listed in its set of parentheses. We still must however list that empty set of parentheses there to represent v_7 for if it was not included our encoded string would not match the actual graph it was designed to represent.

Our last encoding scheme that we will look at is the *adjaceny matrix rows* encoding scheme also presented by Garey and Johnson. We can take any graph with n vertices and change it into a matrix of n *x* n size, where rows 1 through n represent vertex 1 through n and columns 1 through n also represent vertices 1 through n. So for Figure **??** it's adjacency matrix would look like:

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 2.1: Adjacency Matrix for the graph in Figure **??**.

To read this matrix, start by examining row 1, representing vertex 1, and determine in what columns there are 1's. In row 1 both columns 2 and 5 have have 1's so this tells us that vertex 1 is adjacent to vertices 2 and 5. If you look at row 2 (vertex 2), the adjacency matrix tells us that vertex 2 shares an edge with vertices 1, 3 and 5. One could continue reading the adjacency matrix in this manner until they reach the last row. At this point you should know that you have 7 vertices with 7 edges and if the graph was encoded correctly, you should be able to reproduce the original graph.

These three encoding schemes are just examples of the infinitely different encoding schemes that one could create for a particular decision problem. There is no restriction on what a certain encoding scheme should look like. The only stipulation is that any encoding scheme used be reasonable, meaning that one particular encoding scheme can be converted to another encoding scheme using some polynomial time algorithm. Now that we know what an encoding scheme looks like, we need to look at the language that will be encoded.

A language is needed to transform a decision problem like Π_{IS} into something that can be read by a program on a computational machine. Once the problem has been transformed into an input then we can feed that information into the machine and it will be able to conclude a yes or no decision based on the given input. For the remainder of this section, I will attempt to give a better overall understanding of what a language is.

Before we can determine what a language is we need to start with the components that make up a language. Just as English is the language of the people in the United States, written English would not exist without the Latin alphabet upon which it is based. The English alphabet is comprised of 26 characters ranging from the letters A to Z. We then connect these letters in finite strings known to us as "words" that allow us to communicate with each other. Compare the finite strings of the latin alphabet "cat" and "xzp". One of these strings, "cat", has meaning in our language where the other, "xzp", is meaningless. The language that

we are going to invent is created in the same way. We start out with an **alphabet** which just consists of a finite set of symbols. Garey and Johnson use the symbol Σ to represent an alphabet in their book *Computers and Intractability: A Guide to the Theory of NP-Completeness* and that same symbol will be used throughout the rest of this paper to do the same. From Σ there is a set of all of the finite strings of symbols that we will denote Σ^*. So ideally what will happen is that we will feed into the computational device a string of symbols, then a program built into the device will do some calculations and return an answer to our decision problem, Π, for a given instance.

But we need to pause a second and think about all of the possible strings that make up Σ^*. Σ^* can actually be divided into three subsets. The first subset contains all of the finite strings that are not recognizable by the program on the computational machine. These are just strings of symbols that are meaningless, like "xzp" in the English language. The second subset contains all of the strings of symbols that represent instances for a particular Π in which the answer to the decision problem for the given instance is no. The last subset of Σ^* and the most important to us is the set of all strings that represent encoded instances for a decision problem in which the answer to the decision problem for a given instance is yes. This third set of finite strings encoded using some encoding scheme e, form a language associated with that particular decision problem. This **language**, L, is the collection of all encoded finite strings from Σ^* that correspond to a specific instance of a given decision problem whose answer for that instance is yes.

Languages are generally associated with a particular decision problem and encoding scheme. For example, the language associated with the INDEPENDENT SET (Π_{IS}) decision problem for some encoding scheme e, would be written $L = L[\Pi_{IS}, e]$. Now that we have clearly defined what a language is for a particular decision problem we need to examine what it means for strings to be accepted by a program and how those accepted strings create a recognizable language.

So what does it take for a string to become an "accepted string of symbols" by our program? For a string of symbols to be accepted by a program, the computational device has to read in the string and the program needs to halt in the yes state (i.e. a yes answer to our decision problem for that instance). If the program continues to run forever or ends in the no state, then that string of variables was not accepted by our program. The last thing to mention is that all of the finite strings that are "accepted" by a program create a language that is *recognized* by the program.

DEFINITION 2.1. The language L_M **recognized** by a program M is given by:
$L_M = \{x \in \Sigma^* : M \text{ accepts } x\}$

A decision problem Π is said to be **solved** when a program M halts for all input strings $x \in \Sigma^*$ and $L_M = L[\Pi, e]$. That is, the language accepted by the program is the set of finite strings from Σ^* under encoding scheme e representing an instance I of Π where that instance is in the set Y_Π. So if $x \in \Sigma^* - L_M$ then the answer to the decision problem for that particular instance is no and if $x \in L_M$ then the answer is yes.

2.2 Deterministic Turing Machines

We are talking about using a computer to solve a decision problem. To talk unambigously about the number of steps or opertations it takes a computer to answer a decision problem, we need a precise description of a computer. We could use a description of a desktop computer but that is needlessly complicated. Instead, we will use a simplified computer called a "turing machine", introduced for this purpose by the English mathematician Alan Turing in the 1930s [**?**]. Many of the properties associated with the original turing device are also part of the two modified turing devices that we will be using. We begin by examining the deterministic one-tape turing machine, DTM, which is used to solve the decision problems

in language P. This DTM consists of the tangible objects: a finite state control, a read-write head, and a tape made up of a two-way infinite sequence of tape-squares as well as a program M that runs on the device. The finite state control of a DTM keeps track of what state the program is in. The program is set to begin operation in the initial state q_0. As the program runs the finite state control will continue to update the current state of the program until it reaches the terminal stage where it would end in either q_Y or q_N. If the program ends in the q_Y state then the answer to the decision question is yes, if it ends in the q_N state then the answer is no.

The read-write head on the turing machine has the ability to read/scan into the program the symbol located on the tape as well as to erase the symbol on the tape and then record a new symbol from Γ. Γ is a finite set of tape symbols which include those symbols in Σ, the set of all input symbols, along with a distinguishable blank symbol, b. The actions of the read-write head depends on two things: the current state of the program and the input symbol on the tape. Depending on these two inputs, a transition function δ that is part of the program will then direct the DTM to update the state of the program, either leave the input symbol as it is or erase and record a new symbol on the tape and lastly move the tape either one square to the left or one square to the right. The tape is really the input for the DTM. Located on the tape is a string of symbols arranged in such a way so that each square contains at most one symbol per square. The string of symbols, x, located on the tape are preceded and followed by a blank symbol which notify the program when to start and stop reading the symbols in. The length of x is denoted by $|x|$. We use the notation $|x| = n$ to represent a finite string with n symbols. Now that we know what the components are let's examine the process that a turing machine goes through when it is fed in an input representing some instance of a decision problem.

Let's imagine that we have a DTM with a program M that will help us solve Π_{IS} for any $G = (V, E)$. And suppose further that we want our turing machine to examine the graph

from Figure **??** and tell whether there is a independent set V' with $|V'| \geq 3$. So we would take our graph and encode it to look like a finite string of symbols that could be placed on a tape which will be read by the DTM. We record our encoded string on the tape and proceed to load the tape into the machine so that the read-write head of the machine is above the very first symbol of our string. The machine is turned on and the program begins in the initial state q_0 and the read-write head reads in the first symbol. The transition function γ identifies these two items and then maps it to a triple which consists of an updated state, an updated symbol and a move on the tape. Upon the instructions of γ the finite state control keeps track of the updated state, while the read-write head either erases and records a new symbol on the scanned tape square or leaves the original symbol unchanged. The tape is then moved either one square to the left or one square to the right and the process continues. This process will continue until the program ends either in state q_Y or q_N. Based on what we know about that graph, we know that the program should end in state q_Y since the independence number of the graph in Figure **??** is 4.

The time for which it takes any turing machine to complete this process is very important. Time here is not thought of in how many seconds or minutes it might take for the program to run but as the number of steps that occur during computation until the program has entered it's halt state. Note that the number of steps can depend on the length of the input string for a particular instance of a decision problem. More importantly time is defined by a function which is summarized in the definition below.

DEFINITION 2.2. A **time complexity function** ($T_M : Z^+ \to Z^+$) for a DTM program M that halts for all inputs $x \in \Sigma^*$, is given by:

$T_M(n) = \max\{\text{m: there is an } x \in \Sigma^* \text{with } |x| = n, \text{ such that the computation of } M \text{ on input } x \text{ takes time } m\}$.

So the time complexity function T_M takes an integer n and maps it to m, the largest number of steps it would take to compute the answer to a decision question with input length

n. Time and steps are both positive integers. A program is said to be a *polynomial time DTM program* if there exists a polynomial p such that the time complexity function for an integer n is less than or equal to some polynomial $p(n)$. How does time tie into the whole process? Well if there is a decision problem that can be solved for a given instance under some encoding scheme by a DTM in polynomial time, then that problem is said to belong to the class P of problems.

To solidify the understanding of a deterministic turning machine, let's look at a particular decision problem that we will show to be in P.

EDGE SET Π_{ES}

INSTANCE: Given graph $G(V,E)$ in Figure **??** and the integer 4.

QUESTION: Is there a set E' of edges where $|E'| \geq 4$?

So taking the graph from Figure **??** we know that we can use one of the three encoding schemes that were discussed in the previous section to encode our graph. Let's choose the vertex list, edge list encoding scheme. In doing so, we will list out all the vertices and all the edges for our graph. It takes four symbols to record a vertex and ten symbols to record each edge. Recall that the vertices are listed first, followed by edges listed in pairs inside of parantheses. Our symbols that we will use come from the alphabet $\Sigma = \{[,],(,),v,1,2,3,4,5,6,7,8,9,,\}$. We encode our string onto the tape that will be fed into the turing machine. Since our graph has seven vertices and seven edges it will take 98 symbols to encode our graph on the tape. We will also need to encode our integer on the tape as well to let the program know the number of edges that it is looking for. That brings our string up to 99 symbols. Lastly, a single blank symbol must precede and follow the string of symbols letting the program know where to begin and end reading the input. So the total length of our string is 101, $|x| = 101$.

Our program reads the string in and now needs to do some computation to see if there is a set of edges, E', where $|E'| \geq 4$. So what we really want to know is, is the program on DTM going to halt in the yes state, q_Y, when we feed the input into it. Well we could create a program that upon examining the last symbol of our string, here the symbol "4", will go back and count the number of ")" symbols that precede it. It can store the current number of ")" symbols scanned in it's finite state control and continue to move the tape to the left stopping in state q_Y if the current number of ")" symbols scanned is greater than or equal to the last symbol of our string. For our example, the least amount of steps to check to see if the number of edges is greater than the last symbol is 10 times the last symbols value. This occurs when our graph has at least the same amount of edges required by our question. Since the question asks does our graph have some set of edges whose size is greater than or equal to four, it is not necessary for our program to determine if our graph has greater than 4 edges. What about if our graph has less than four edges? Well the program would continue to check until it hit the blank symbol at the beginning of our string and end in the no state, q_N. The number of steps required here is equal to the length of our input string. Clearly for our graph, the program will end in state q_Y since $|E| \geq 4$.

Is $\Pi_{ES} \in P$? Well, assume we have an input string of length n on our turing machine tape. It will take the machine n steps to read this string. As the machine is reading in our string of length n, the program counts the number of left parentheses symbols in the vertex-edge representation. Thus, it will take a maximum of n steps to compute the answer to our decision question. So, our time complexity function $T_M(n) = n$ which is clearly a polynomial in n, so $\Pi_{ES} \in P$.

2.3 Nondeterministic Turing Machine

Now that DTM's have been briefly explained we need to examine the other turing device, the nondeterministic one tape turing machine. A nondeterministic one-tape turing machine, NDTM, is very similar to the DTM machine except that it has one additional component added to it and one additional stage added to it's program. The NDTM still has a finite state control and a read-write head but is equipped with an additional guessing module that has it's own write-only head. Just like with the original DTM we would record our given instance of a decision problem on the tape one symbol at a time. We then load the tape into the machine so that the read-write head is lined up with the first symbol in our finite string of input symbols. Once this has occured our initial stage, commonly referred to as the guessing stage of the program, can begin. With the first symbol lined up with the read-write head, the guessing head (write-only) becomes activated and will record a string of symbols from the tape alphabet Γ and the read-write head will enter the "inactive" state.

During the guessing stage there are a couple of possiblities that can occur. The first is that the guessing head continuously records symbols and never halts. If this occurs the read only head will remain inactive and our program will never run, which means the program will not answer our decision problem. The second possibility is that the guessing head enters a finite string of symbols that is essentially a mixture of recognizable input symbols and some garbage symbols not recognizable by our program. The third possibility is that the guess only head enters a finite string of symbols that are recognizable by the program and will eventually lead the program to a halting state, either q_Y or q_N. Once the guessing module has finished recording it's symbols on the tape, the guessing head becomes inactive and the read-write head is activated.

The program for the NDTM works as follows. The read-write head reads in the string of symbols that the guessing head has recorded. After each symbol has been read, just like

in the DTM, the transition function γ identifies the current state and the read symbol and transitions the program to an updated state, an update (change/leave alone) of the current symbol on the tape and a move of the tape one square to the right or to the left. The program continues until the finite state control has reached a halting state, either q_Y or q_N. Because the guessing head has the ability to produce an infinite number of possible finite strings of tape symbols we need to separate these strings into two sets of easily identifiable strings. The computation that occurs by the program when a guessed string of tape symbols along with the encoded strings of specific instances of a given decision problem are read in which the program ends in state q_Y is called an *accepting computation.* Further, computation on the remaining guessed strings and encoded instances, where the NDTM program does not halt or ends in state q_N are called non-accepting computations. Again, just as with the DTM, the collection of strings in which the program ends in the yes state creates the language that is "recognized" by the program. So the language for program M, L_M, is defined to be the set of all finite strings (guess and instance) from Σ^* where the computation by program M on the string ends in state q_Y.

Again we will measure computation time in terms of the number of steps it takes to run through the computation program. Notice however that the number of steps this time includes the number of steps that it takes for the guessing head to record the guess and the number of steps that it takes the program to run through and check that instance. We again have a time complexity function ($T_M : Z^+ \to Z^+$) but is now defined to be:

$$T_M(n) = \max(\{1\} \cup \{\text{m: there is an } x \in L_M \text{ with } |x| = n \text{ such that the time to accept } x \text{ by } M \text{ is } m\}).$$

So essentially we have two options. For accepting computations on M, we set time equal to m, where m is the number of steps it takes for program M to run on a string of size n through the guessing and checking stages and ultimately ending in state q_Y. For all other

strings of size n that yield non-accepting computations we set time equal to 1. We say that NDTM program M is a polynomial time NDTM program if there exists some polynomial p, such that the time complexity function $T_M(n)$ is less than or equal to $p(n)$ for all inputs of length one or higher. Therefore, any decision problem that can be solved for a given instance under some encoding scheme by a NDTM in polynomial time, then that problem is said to belong to the class NP of problems. Lastly, any problem that belongs to the class P of problems also belongs to the class NP, since if they were able to be solved by a DTM we can get rid of the guessing stage in the NDTM and just check the initial input from the decision problem.

Polynomial Transformations and SAT

Our main goal is to prove that a decision problem, Π_{IS}, is NP-complete. In order to show that any decision problem is NP-complete you need to show two things. The first item that needs to be shown is that $\Pi \in$ NP. This means that our decision problem Π can be solved by a nondeterministic turing machine in polynomial time. Therefore we must be able to feed in our input, in the case of the Π_{IS} an encoded graph and the NDTM should be able to guess some set of vertices and check to see whether or not they are independent and greater than some given integer K. Secondly, one needs to take an existing known NP-complete problem $\acute{\Pi}$ and show that there is a polynomial transformation function from $\acute{\Pi}$ to Π. If both of these conditions can be shown then Π is an NP-complete problem. In this chapter we begin by exploring the idea of polynomial transformations. We will discuss what a polynomial transformation is and then examine some important properties that will allow us to transform one NP decision problem into NP decision problem. After polynomial transformations have been covered we will end the chapter by an in-depth examination of the first NP-complete problem, proved by Cook in [**?**] in 1971.

3.1 Polynomial Transformations

If there were a process that would allow us to take a decision problem and change it into another decision problem, it would open us up to the possibility that if we found a solution to one of those decision problems we would also have found a solution to the other. On a larger scale, what if we had a set of decision problems where we could transform each problem in

the set to any other problem in the set through a series of steps. Ideally, if this were the case we could spend all of our time and effort trying to find a solution to one decision problem instead of the collective group. If we were able to solve this problem, essentially we would be able to solve the rest of the problems in the set. This transformation idea is essential to the process that will allow us to convert decision problems into other decision problems.

To begin this transformation we need to start with two alphabets Σ_1 and Σ_2. Remembering from before, an alphabet was defined to be a finite set of symbols. From Σ_1, we form a subset Σ_1^*, that is the set of all finite strings of symbols from Σ_1. Likewise, Σ_2^* is created in the same manner from Σ_2. A language, $L_1 \subseteq \Sigma_1^*$, is the set of all finite strings from Σ_1^* that will provide a yes answer to the decision problem using Σ_1 as it's alphabet. Likewise, L_2 is a language created from Σ_2^*. Now that we have defined the two alphabets, two sets of finite strings from those alphabets and two languages, we are now ready to define our transformation from one set of finite strings of one alphabet to a set of finite strings from another alphabet.

DEFINITION 3.1. A **polynomial transformation** from a language $L_1 \subseteq \Sigma_1^*$ to a language $L_2 \subseteq \Sigma_2^*$ is a function $f : \Sigma_1^* \to \Sigma_2^*$ that satisfies the following two conditions:

1. There is a program M for some DTM that computes f in polynomial time.
2. For all $x \in \Sigma_1^*$, $x \in L_1$ if and only if $f(x) \in L_2$.

If those two conditions are satisfied, then we say that L_1 transforms to L_2 and abbreviate it $L_1 \propto L_2$. This setup of polynomial transformations leads us to some very important results. The first of which is given below followed by it's proof:

LEMMA 3.2. If $L_1 \propto L_2$, then $L_2 \in P \Rightarrow L_1 \in P$.

Proof. Assume that $L_1 \propto L_2$ and $L_2 \in P$. Then there must exist a function $f : \Sigma_1^* \to \Sigma_2^*$ that is a polynomial transformation from L_1 to L_2. Let M_f be a DTM program that computes f. Since $L_2 \in P$, there must exist some polynomial time DTM program that recognizes L_2,

let's call it M_2. We can create a DTM program for recognizing L_1 by composing program M_f with M_2. Now for any input $x \in \Sigma_1^*$ we apply the program M_f which outputs $f(x) \in \Sigma_2^*$. We then input $f(x)$ into program M_2 to determine if $f(x) \in L_2$. Since $x \in L_1$ if and only if $f(x) \in L_2$ by the definition of polynomial transformation, we have a program $M_2 \circ M_f$ that recognizes L_1. It is easily seen that this program must operate in polynomial time since the running time of both M_f and M_2 are bounded by polynomial functions and adding two polynomial functions produces another polynomial function. Thus $M_2 \circ M_f$ operates in polynomial time. □

Now that we have seen a polynomial transformation from one language to another language, the idea can be extended to decision problems. Let Π_1 and Π_2 be decision problems, with encoding schemes e_1 and e_2, then $\Pi_1 \propto \Pi_2$ if there exists a polynomial transformation from $L[\Pi_1, e_1]$ to $L[\Pi_2, e_2]$. Moving away from the idea that one language transforms into another language toward the broader idea that one decision problem transforms into another we see that there exists a polynomial transformation function $f : \Pi_1 \to \Pi_2$ if it satisfies the following conditions. The first condition is that f is computable by a polynomial time algorithm. The second condition is that for all instances of decision problem Π_1, an instance I is in the yes instances for Π_1 if and only if $f(I)$ is in the yes instances for Π_2. Extending polynomial transformation to decision problems will help us later on transform the SATISFIABILITY decision problem to the 3SAT decision problem and then take 3SAT and transform it to the INDEPENDENT SET problem.

It should be noted that polynomial transformations have the transitive property that if $L_1 \propto L_2$ and $L_2 \propto L_3$ then $L_1 \propto L_3$. This can easily be shown by composition of polynomial transformation functions. If $f_1 : \Sigma_1^* \to \Sigma_2^*$ is a polynomial transformation from L_1 to L_2 and $f_2 : \Sigma_2^* \to \Sigma_3^*$ is a polynomial transformation from L_2 to L_3 then we can create a function $f : \Sigma_1^* \to \Sigma_3^*$ by defining $f(x) = f_2(f_1(x))$ for all $x \in \Sigma_1^*$. This is the desired transformation

from L_1 to L_3, since $f(x) \in L_3$ if and only if $x \in L_1$. It follows that f operates in polynomial time since f_1 and f_2 can be computed by a polynomial time DTM program.

These two properties of polynomial transformations allow us to define what it means for a language to be NP-complete. A language L is defined to be **NP-complete** if $L \in NP$ and, for all other languages $L' \in NP$, $L' \propto L$. We then transfer this idea to decision problems saying that decision problem Π is NP-complete if decision problem $\Pi \in NP$ and, for all other decision problems $\Pi' \in NP$, $\Pi' \propto \Pi$. This idea allows us to see that if any NP-complete decision problem can be solved in polynomial time, then all problems in NP can be solved in polynomial time. Also if any NP-complete problem is shown to be intractable, have an exponential time complexity function, then all other NP-complete problems are intractable. The idea of polynomial transformability and NP-completeness provide us with our second lemma which will be useful in the following section to show that the SATISFIABILITY Decision Problem is NP-complete.

LEMMA 3.3. If L_1 and L_2 belong to NP, L_1 is NP-complete, and $L_1 \propto L_2$, then L_2 is NP-complete.

Proof. Since L_2 is NP, all that is left to show is that, for every other language in NP, that language transforms to L_2. Consider any other language in NP, say L'. Since L_1 is NP-complete, by defintion L' must transform to L_1. Since polynomial transformations are transitive, and $L' \propto L_1$ and $L_1 \propto L_2$ then $L' \propto L_2$. □

We can extend this lemma to decision problems since they are based on languages and encoding schemes. If Π_1 and Π_2 belong to the set of NP decision problems, and Π_1 is NP-complete and $\Pi_1 \propto \Pi_2$ then Π_2 is NP-complete. We will use this throughout the remaining chapters to prove that certain decision problems are NP-complete. However, for this to work we need to have at least one NP-complete decision problem. In the next section, we look at the decision problem that was first proved to be NP-complete.

3.2 The SATISFIABILITY Decision Problem

The first decision problem that was shown to be NP-complete is the Satisfiability Problem, SAT for short. In 1971, Cook proved in [**?**] that SAT was NP-complete. The following year, Karp proved that there were 21 more decision problems that were NP-complete in [**?**]. The SATISFIABILITY problem is defined as follows:

SATISFIABILITY Π_{SAT}
INSTANCE: A set U of variables and a collection C of clauses over U.
QUESTION: Is there a satisfying truth assignment for C?

Before we begin, it is necessary to provide some background information on Boolean logic upon which Π_{SAT} is based. We start with a few definitions. A set U of boolean variables is defined as $U = \{u_1, u_2, ..., u_m\}$. There is a function t from the set of variables, U, to the set of truth values $\{T, F\}$ where $t(u) = T$ if u is true under t and $t(u) = F$ implies u is false under t. The variables u and $\bar{u}$ are **literals** over U if $u \in U$. Think of $\bar{u}$ as the negation of u so it will take the opposite truth value under t. For example, the literal u is said to be true if and only if $t(u) = T$. Also, the literal $\bar{u}$ is associated with true if and only if $t(u) = F$. A **clause** over U is a collection of literals that forms a disjunction. This clause of literals is said to be satisfied if one or more of the literals is true under truth assignment t. A collection of clauses, C, over U is said to be **satisfiable** if and only if there exists some truth assignment for U that satisfies all the clauses in C simultaneously. That truth assignment is called a "satisfying truth assignment" for C.

Let's look at an example. Let $U = \{u_1, u_2, u_3, u_4, u_5\}$ and let $C = \{\{u_1, u_3\}, \{\bar{u_2}, \bar{u_3}\}, \{\bar{u_1}, u_4, u_5\}\}$. A satisfying truth assignment for C would be $t(u_1) = F, t(u_2) = F, t(u_3) = T, t(u_4) = T, t(u_5) = F$. In this truth assignment, we can see that $t(C) = \{\{F, T\}, \{T, F\}, \{T, F, T\}\}$

thus satisfying the condition that every clause has at least one literal assigned to true. Notice that if we changed only $t(u_3) = F$ and kept all other assignments the same notice the function would no longer be a satisfying truth assignment.

Now that some background information has been provided, we are ready to show that SAT is NP-complete. We start by stating Cook's theorem and then work through his proof providing further explanation than is found in both [**?**] and [**?**].

THEOREM 3.4. SATISFIABILITY is NP-complete.

Proof. To prove that SAT is NP-complete we need to first show that SAT is in NP. Secondly, we must show that every instance of a problem Π in NP can be transformed to an instance of SAT. Showing that SAT is in NP is relatively easy. All we need to do is to have some program on a non-deterministic turing machine guess a truth assignment for the given variables and then checks to see if that truth assignment satisfies all of the clauses for the given instance of SAT. This program can be accomplished in polynomial time. Thus, the first condition has been satisfied.

Showing that the second condition is true is going to be more difficult. First, there are not any known NP-complete problems, so we are going to have to work at transforming languages in NP than transforming problems in NP. We will start by representing the decision problem SAT as a language, L, combined with some reasonable encoding scheme, e. More formally we define the language for SAT to be $L_{SAT} = L[SAT, e]$. We need to show that for any language in NP, that language transforms to the language for SAT, i.e. ($L \in NP$, $L \propto L_{SAT}$). Since there are infinitely many languages in NP, it would be impossible to create a proof showing a transformation from each individual language to L_{SAT}. Therefore, we need to describe all the languages in NP in a standard way, by associating a polynomial time NDTM with each language. This allows us to work with a generic polynomial time NDTM

program and to create a generic transformation from the language that NDTM recognizes to L_{SAT}.

Let M be a polynomial time NDTM program, specified by: a finite set of tape symbols, Γ, an input alphabet, Σ, a blank symbol, b, a finite set of states, Q, an initial state, q_0, a terminal state with a "no" answer, q_N, a terminal state with a "yes" answer, q_Y and lastly a transition function, δ. The time complexity function for program is T_M, where $T_M(n) \leq p(n)$ for all $n \in Z^+$. So our program operates in polynomial time for all input strings x of size n. Our generic transformation function f_L will be created in terms of $M, \Gamma, \Sigma, b, Q, q_0, q_N, q_Y, \delta$, and p. Essentially what we will be doing is taking a string of variables, x, from L and mapping it to an instance, represented by a set of clauses, in SAT by the transformation f_L. Cook defines this transformation f_L to have the property that for all $x \in \Sigma^*$, $x \in L$ if and only if $f_L(x)$ has a satisfying truth assignment. This means that some finite string of variables from the alphabet x is accepted by M if and only if the instance x is mapped to, $f_L(x)$ is a satisfiable instance of SAT.

If the input string x is accepted by program M, then we know that there is an accepting computation for M on x such that both the number of steps in the checking stage and the number of symbols in the guessed string are finite and thus bounded by some polynomial $p(n)$ where n is the length of string x. The tape squares used can only be those numbered between squares $-p(n)$ through $p(n)+1$, since the read-write head of the NDTM begins at tape square 1 and moves at most one square to the left or right after each step.

At this point, let's define Q, the set of states for our progam M to be $Q = \{q_0, q_1 = q_Y, q_2 = q_N, ..., q_r\}$ where $r = |Q| - 1$. Also, let the set of input symbols, Γ, be defined as $\Gamma = \{s_0 = b, s_1, s_2, ..., s_v\}$ where $v = |\Gamma| - 1$. The computation process of x on M allows us to create variables for our instance in SAT, $f_L(x)$, based on the contents of the square being scanned, the current state our program is in, and the current position of the read-write head. All of these items will depend on what step the program is on in the computation process.

Variable	Range
$Q[i,k]$	$0 \leq i \leq p(n)$ and $0 \leq k \leq r$
$H[i,j]$	$0 \leq i \leq p(n)$ and $-p(n) \leq j \leq p(n)+1$
$S[i,j,k]$	$0 \leq i \leq p(n)$ and $-p(n) \leq j \leq p(n)+1$ and $0 \leq k \leq v$

Figure 3.1: Variables in $f_L(x)$ and the ranges of their elements

There will be three types of variables derived from the computation process of the accepted string x on M: $Q[i,k]$, $H[i,j]$ and $S[i,j,k]$. They are defined as follows:

Variables of the type $Q[i,k]$ are those that are associated with the state of the program during the computation process of input string x. They can be interpreted as "at time i, program M is in state q_k". Since our checking computation takes at most $p(n)$ time, we bound i from the intial time $i = 0$ to the time needed to end the computation in the yes state $p(n)$. The k element of our variable represents the state of the program during computation is bounded by 0, which is q_0 the initial state, and q_r, the highest possible state.

Variables of the type $H[i,j]$ are those associated with the position of the read-write head of our NDTM. They are to be interpreted as "at time i, the read-write head is scanning tape square j". Again i, the time needed to complete the computation by M, is bounded by 0 and $p(n)$. The j element of our variable is associated with the tape square the read-write head is located above. Since our input string x and the guessed string w are finite they are able to be bounded by $-p(n)$ and $p(n)+1$, where $p(n)$ is the number of steps in the checking stage and the number of symbols in the guessed string . Since our read-write head is positioned over the square located in position 1 of our turing tape, we know that it can go a maximum of $p(n)$ squares to the right before it is over a blank symbol. This will occur at square $p(n)+1$. Since the number of steps in the checking stage is at most $p(n)$, the read write-head can go at most $p(n)$ squares in the direction to the left from the square located in position 1 on our turing tape.

The last type of variables, $S[i,j,k]$, are those that are associated with the symbols located on the tape. These variables are interpreted as "at time i, the contents of tape square j is symbol s_k". Both i and j for this variable are defined in the same way that they were defined in variables of the type $H[i,j]$. Element k of $S[i,j,k]$ is referencing the symbol located in the square j at time i. This symbol s_k must belong to the set: $\{s_0, s_1, s_2, ..., s_v\}$.

Now that variables constructed by f_L have been explained, let's take a second to examine their creation process. A string $x \in \Sigma^*$ and a guessed string w of length $p(n)$ is fed into a NDTM and a program M performs a checking computation. If M accepts string x and its guess then we know that x has an accepting computation on M. Therefore $x \in L_M$ and it's computation on M takes $p(n)$ or fewer steps. The variables in $f_L(x)$, $Q[i,k], H[i,j], S[i,j,k]$, are created from the computation of x on M. Now, we just need to show that the collection of clauses $f_L(x)$ are satisfied by some truth assignment. The way that we do this is that we divide the clauses in $f_L(x)$ into six groups. Garey and Johnson defined these groups on page 43 in [**?**] as in Figure **??**.

The groups of clauses in $f_L(x)$ seen in Figure **??** can be explained in the following manner. Each group of clauses will be presented one at a time along with justification of why each group of clauses would be satisfied when x is part of the language recognized by M, i.e. $x \in L_M$. Clause Group G_1 is comprised of the clause $\{Q[i,0], Q[i,1], ..., Q[i,r]\}$ and the remaining clauses that occur in pairs and take the form $\{\overline{Q[i,j]}, \overline{Q[i,j']}\}$. All of the clauses in group G_1 are linked the state of program M at some time i in the checking process. The clause $\{Q[i,0], Q[i,1], ...Q[i,r]\}$ will be satisfied if at time i in the checking stage program M is in some state in Q where $Q = \{q_0, q_1 = q_Y, q_2 = q_N, ..., q_r\}$. This will occur if $x \in L_M$. The remaining clauses in G_1 of the form $\{\overline{Q[i,j]}, \overline{Q[i,j']}\}$ will be satisfied if one of these literals in mapped to true by a truth assignment function. Before going further it might be helpful to the reader if we give a interpretation of $\overline{Q[i,j]}$. $\overline{Q[i,j]}$ should be read as "at time i, M is not in state j". Consider the following scenarios and see how no matter what

Clause Group	Clauses in group
G_1	$\{Q[i,0], Q[i,1], ...Q[i,r]\}$, $0 \leq i \leq p(n)$ $\{\overline{Q[i,j]}, \overline{Q[i,j']}\}$, $0 \leq i \leq p(n)$, $0 \leq j < j' \leq r$
G_2	$\{H[i,-p(n)], H[i,-p(n)+1], ..., H[i,p(n)+1]\}$, $0 \leq i \leq p(n)$ $\{\overline{H[i,j]}, \overline{H[i,j']}\}$, $0 \leq i \leq p(n)$, $-p(n) \leq j < j' \leq p(n)+1$
G_3	$\{S[i,j,0], S[i,j,1], ..., S[i,j,v]\}$, $0 \leq i \leq p(n)$, $-p(n) \leq j \leq p(n)+1$ $\{\overline{S[i,j,k]}, \overline{S[i,j,k']}\}$, $0 \leq i \leq p(n)$, $-p(n) \leq j \leq p(n)+1$, $0 \leq k < k' \leq v$
G_4	$\{Q[0,0]\}$,$\{H[0,1]\}$, $\{S[0,0,0]\}$, $\{S[0,l,k_1]\}$, $\{S[0,2,k_2]\}$,...,$\{S[0,n,k_n]\}$, $\{S[0,n+1,0]\}$, $\{S[0,n+2,0]\}$,..., $\{S[0,p(n)+1,0]\}$, where $x = s_{k1}s_{k2}...s_{kn}$
G_5	$\{Q[p(n),1]\}$
G_6	$\{\overline{H[i,j]}, \overline{Q[i,k]}, \overline{S[i,j,l]}, H[i+1,j+\Delta]\}$ $\{\overline{H[i,j]}, \overline{Q[i,k]}, \overline{S[i,j,l]}, Q[i+1,k']\}$ $\{\overline{H[i,j]}, \overline{Q[i,k]}, \overline{S[i,j,l]}, H[i+1,j,l']\}$ with $0 \leq ß < p(n)$, $-p(n) \leq j \leq p(n)+1$, $0 \leq k \leq r$ and $0 \leq l \leq v$ for all clauses. if $q_k \in Q - \{q_Y, q_N\}$, then Δ, k', and l' are such that $\delta(q_k, s_l) = (q_{k'}, s_{l'}, \Delta)$ if $q_k \in \{q_Y, q_N\}$, then $\Delta = 0$, $k' = k$, and $l' = l$.

Figure 3.2: The six clause groups in $f_L(x)$

state M is in at least one of the literals must be mapped to true. If at time i, M is in state j then $t(\overline{Q[i,j]}) = F$ and $t(\overline{Q[i,j']}) = T$. If at time i, M is in state j' then $t(\overline{Q[i,j]}) = T$ and $t(\overline{Q[i,j']}) = F$. Lastly, if at time i, M is in some state other than j or j' then $t(\overline{Q[i,j]}) = T$ and $t(\overline{Q[i,j']}) = T$. So, as long as $x \in L_M$, clauses in G_1 will be satisfied.

Clause group 2, G_2 consists of the clause $\{H[i,-p(n)], H[i,-p(n)+1], ..., H[i,p(n)+1]\}$ and the remaining clauses of the form $\{\overline{H[i,j]}, \overline{H[i,j']}\}$. The first clause of G_2 is satisfied if at time i in the checking stage, the read-write head of the NDTM is scanning a tape square on the turing tape that is part of the original string x, the guessed string w or the blank symbol that is located between them at square location 0. The remaining clauses of the form $\{\overline{H[i,j]}, \overline{H[i,j']}\}$ will be satisfied if the read-write head is scanning exactly one tape square either in the squares on the tape representing x, w or the blank symbol at location 0. In order to show this is true, consider the following scenarios. At time i, the read-write head

is scanning square j. Well, $\overline{H[i,j]}$ is clearly false since $\overline{H[i,j]}$ is interpretted as "at time i, the read-write head is not scanning tape square j. Meanwhile it's pair in the clause $\overline{H[i,j']}$ is true since $\overline{H[i,j]}$ is interpreted as "at time i, the read-write head is not scanning tape square j'" and by definition $j \neq j'$. If we continue the same logical arguement from the clauses in group G_1 it is easy to see that we will have at least one literal mapped to true in every pair of clauses of the form $\{\overline{H[i,j]}, \overline{H[i,j']}\}$ in G_2. Again, as long as $x \in L_M$, clauses in G_2 will be satisfied.

Clause group 3, G_3 is made up by the clause $\{S[i,j,0], S[i,j,1], ..., S[i,j,v]\}$ and those pairs of clauses that take the form $\{\overline{S[i,j,k]}, \overline{S[i,j,k']}\}$. These clauses correspond to the symbol located on tape square j at time i. The first clause $\{S[i,j,0], S[i,j,1], ..., S[i,j,v]\}$ will be satisfied if at some time i, for square j the read-write head is scanning in a symbol from Γ our set of input symbols. This will be true if x is accepted by M. The remaining clauses in G_3 occur in pairs and are satisfied as long as there is exactly one input symbol per tape square. $\overline{S[i,j,k]}$ is interpreted as " at time i, the contents of square j is not symbol s_k". Well this will be true if the symbol in the square is any other symbol from Γ other than s_k. Likewise, $\overline{S[i,j,k']}$ is interpreted as " at time i, the contents of square j is not symbol s'_k". This will be true as long as the symbol in the square is any other symbol from Γ other than s'_k. Since k and k' are bounded by 0 and v, the pairs will always contain at least one literal mapped to true. In fact, the pairs will have both literals mapped to true as long as the symbol in square j at time i is any other symbol in Γ other than s_k or s'_k. Here again, as long as $x \in L_M$, the clauses in G_3 will be satisfied.

The first three clauses in G_4 are $\{Q[0,0]\}$,$\{H[0,1]\}$, $\{S[0,0,0]\}$. $\{Q[0,0]\}$, $\{H[0,1]\}$ and $\{S[0,0,0]\}$ all pertain to when $i = 0$, that is , when program M is in the initial state q_0. As long as at time $i = 0$ program M is in the intial state q_0 and the read-write head is scanning the tape square located at square 1 of the turing tape and the contents of the square located at 0 on the turing tape has the blank symbol in it, the three clauses $\{Q[0,0]\}$

,$\{H[0,1]\}$, $\{S[0,0,0]\}$ will be satisfied. The remaining clauses in G_4 are satisfied if at time $i = 0$ the finite input string x of size n is listed so that each individual symbol in the string x is listed one symbol per square on the turing tape starting with the first symbol of the string in tape square 1 and the last symbol of the string in tape square n. All other tape squares on the turing tape past square n will contain the blank symbol s_0. Since the input string x was accepted by program M and was of finite size, $|x| = n$, before program M begins the turing tape must contain the string x listed out symbol by symbol from square 1 to square n on the turing tape with all squares past n blank. Since the program hasn't started yet, no symbols on the tape will have been changed, thus the clauses in G_4 will be satisfied.

The single clause in G_5 is interpreted as " at time $p(n)$, M is in state $q_1 = q_Y$". Well, if x was accepted by program M, then the checking computation must occur in $p(n)$ or fewer steps and since M accepts x that implies that the terminal state of the program will be q_Y. So if x is accepted by M, clause group G_5 must be satisfied.

The last group G_6 contains three clauses and we will look at each one individually. It should be noted that these clauses are created based on the transition function δ for M. We can interpret the first clause, $\{\overline{H[i,j]}, \overline{Q[i,k]}, \overline{S[i,j,l]}, H[i+1, j+\Delta]\}$, as follows:

$\overline{H[i,j]}$ tells us that "At time i, the read-write head is not scanning tape square j".
$\overline{Q[i,k]}$ tells us that "At time i, program M is not is state q_k".
$\overline{S[i,j,l]}$ tells us that "At time i, the contents of tape square j is not symbol s_k".
$H[i+1, j+\Delta]\}$ tells us that at time $i+1$, the read-write head is scanning square $j+\Delta$.

Consider the following case, at time i, the read write head is scanning tape square j which contains symbol s_k and the progam is in state q_k. It is easy to see that the truth assignments for clause one would be $\{F,F,F,T\}$. If any of the elements were to change say at time i the read-write head is scanning tape square j which contains symbol s'_k and

the program is in state q_k. The truth assignments for the clause would be $\{F, F, T, T\}$. No matter at time i what square the read-write head is scanning, the state of the program, or the contents of the tape square being scanned, at least one of the literals in the first clause will be mapped to true. The same argument can be made for the following two clauses in G_6, thus the clauses in G_6 will be satisfied if the transition function δ is followed.

The following clauses were constructed in a way so that if we input a string x and a guessed string w so that it is a recognized input for program M in at most $p(n)$ steps then this computation on M will determine a set of true and false values for all of the variables in the six groups of clauses in $f_L(x)$. The assignment of these true and false values will be constructed in such a way that if $x \in L_M$ then all clauses in $f_L(x)$ are simultaneously satisfied.

□

SAT to 3SAT

For the decision problem 3SAT, Π_{3SAT}, we have an instance that is made up a collection of clauses in which each clause contains exactly three literals. In reality, it is just a restricted version of Π_{SAT} where each clause contains exactly three literals. A more exact description is as follows:

3-SATISFIABILITY Π_{3SAT}

INSTANCE: A collection $C = \{c_1, c_2, ..., c_m\}$ of clauses of a finite set U of variables such that $|c_i| = 3$ for $1 \leq i \leq m$.

QUESTION: Is there a satisfying truth assignment for U that satisfies all the clauses in C?

Now that we have given a description of Π_{3SAT} we need to show that it is in the set of decision problems that are NP-complete. We can do this by showing that Π_{3SAT} is in NP and then constructing a polynomial transformation from Π_{SAT} to Π_{3SAT}. The first of these two steps is quite quick. Π_{3SAT} is in the set of NP decision problems since a nondeterministic turing machine need only guess a truth assignment for the variables and check in polynomial time (at most $3m$ steps) whether that truth setting satisfies all the three-literal clauses. Showing that there is a polynomial transformation from Π_{SAT} to Π_{3SAT} takes a bit more time and effort. We begin the setup for the construction of this transformation by examining the pieces that make up an instance for each decision problem. Note that the setup and transformation that we are using is presented in [**?**]. It is my intention to provide a

more detailed explanation of the setup and truth assignment than is given in the book.

We begin by letting $U = \{u_1, u_2, ..., u_n\}$ be a set of variables and $C = \{c_1, c_2, ..., c_m\}$ be a set of clauses making up some instance for SAT. We need to construct a collection of three-literal clauses C' on a set of variables U' so that C' is satisfiable if and only if C is satisfiable. C' will be a collection of clauses created from the original variables U from the SAT problem along with some additional variables from a new set, ${U_j}'$. The use of variables from ${U_j}'$ are restricted to only clauses in ${C_j}'$. Each clause in ${C_j}'$ is created to be equivalent to each $c_i \in C$. This means for every $c_i \in C$ there is a related clause or set of clauses in C' that consist of literals in c_i combined with other new variables from U'_j. If the original instance of SAT is satisfied, the c_i must be satisfied and every clause in C' that relates to c_i will be satisfied. The sets U' and C' are defined as follows:

$$U' = U \cup \left(\bigcup_{j=1}^{m} {U_j}' \right) \text{ and } C' = \bigcup_{j=1}^{m} {C_j}'$$

So U' is a set of variables that contains both the original variables used in SAT along with some new variables that will only be used in 3SAT clauses. Similarly, C' is a set of clauses, where the clauses are made up of variables in the original clauses along with some new variables not in clauses from C. Let some clause c_j from C be given by $\{z_1, z_2, ..., z_k\}$ where the z_i's are all literals derived from the variables in U. The way in which ${C_j}'$ and ${U_j}'$ are formed depends on the value of k, which is the number of literals in a particular clause from C. Since the clauses in 3SAT contain exactly three literals, the creation of the sets ${C_j}'$ and ${U_j}'$ totally depend on the number of literals in each c_j. Since we are transforming Π_{SAT} to Π_{3SAT} we need to look at the number of literals in each clause $c_i \in C$. We will have four cases to look at. These four cases will depend on the number of literals in some c_i from C. The cases that need to be examined are when $|c_i| = 1$, $|c_i| = 2$, $|c_i| = 3$, and $|c_i| > 3$. Garey and Johnson present the following transformation setup based upon those four cases.

Case 1. $k = 1$. $U_j' = \{y_j^1, y_j^2\}$, and $C_j' = \{\{z_1, y_j^1, y_j^2\}, \{z_1, y_j^1, \bar{y}_j^2\}, \{z_1, \bar{y}_j^1, y_j^2\}, \{z_1, \bar{y}_j^1, \bar{y}_j^2\}\}$

Case 2. $k = 2$. $U_j' = \{y_j^1\}$, and $C_j' = \{\{z_1, z_2, y_j^1\}, \{z_1, z_2, \bar{y}_j^1\}\}$

Case 3. $k = 3$. $U_j' = \{\oslash\}$, and $C_j' = \{\{c_j\}\}$

Case 4. $k > 3$. $U_j' = \{y_j^i : 1 \leq i \leq k-3\}$, and $C_j' = \{\{z_1, z_2, y_j^1\}\} \cup \{\{\bar{y}_j^i, z_{i+2}, y_j^{i+1}\} : \{1 \leq i \leq k-4\} \cup \{\{\bar{y}_j^{k-3}, z_{k-1}, z_k\}\}$

Now that the four cases have been presented, in order to show that the constructed set of clauses representing an instance of 3SAT is satisfiable if and only if the original set of clauses C representing an instance of SAT is satisfiable, we will assume that there is a truth assignment $t : U \to \{T, F\}$ that satisfies C.

For the first case when $|c_i| = 1$, we know that there is exactly one literal, z_1 that makes up c_i. Since t satisfies C and $\{z_1\} = c_i \in C$, it must be true that $t(z_1) = T$. Which means the truth assignment function t will satisfy any clause in C_j'. So for case 1, C_j' is satisfied if C is satisfied.

The same argument can be made for case 2. Clauses in case 2 are based off of clauses of size two in C. So $\{z_1, z_2\} = c_i \in C$. If t satisfies C then either $t(z_1) = T$ or $t(z_2) = T$ or both variables are mapped to True under t. Either way, both clauses that make up C_j' will be satisfied by t regardless of the truth value assigned to y_j^1. Thus under case 2, C_j' is satisfied if C is satisfied.

In case three, the clause $c_j \in C_j'$ is the exact same clause $c_j \in C$ so if t satisfies C then some literal in the clause c_j must be mapped to true. Since this is true, C_j' is satisfied if C is satisfied.

The last case is the most complicated one and this occurs where some clause $c_j \in C$ has

more than three literals. If this occurs, we will need to make numerous 3SAT clauses to represent the original clause from SAT. The way that our truth assignment is determined here is not arbitrary anymore. We know that if the original instance from SAT is satisfiable then at least one literal say $z_l \in c_j$ must be mapped to true by the truth assignment function t. We will combine that truth function with another truth assignment function $t' : U' \to \{T, F\}$ which maps some y_j^i to either True or False. This truth function, t' will be an extension of t. The function t' will be defined to be the same as t for variables in U, and remains to be defined for the "new" variables in $U' \setminus U$. Consider the following scenarios and their truth assignment functions t'.

Scenario 1:

If $l = 1$ or 2, that is if z_l is either z_1 or z_2 then we will set $t'(y_j^i) = F$ for $1 \leq i \leq k-3$. If this is the case then the first clause $\{z_1, z_2, y_j^1\}$ in C_j' is satisfied since either z_1 or z_2 is mapped to true by t. In the middle clauses, $\{\{\bar{y}_j^i, z_{i+2}, y_j^{i+1}\} : 1 \leq i \leq k-4\}$, the negated literal, $\bar{y}_j^i$, will be assigned to T by t' when i is greater than or equal to 1 and less than or equal to $k-3$ which holds for all of the middle clauses. Therefore, since each middle clause contains one T, they must be satisfied. Well, by our definition of t' we know $t'(y_j^{k-3}) = F$ so $t'(\bar{y}_j^{k-3}) = T$ and so the last clause is satisfied. So t' is a satisfying truth assignment if $l = 1$ or 2.

Scenario 2:

If $l = k-1$ or k, then set $t'(y_j^i) = T$ for $1 \leq i \leq k-3$. So, in the first clause in C_j' we know $t'(y_j^1) = T$ by truth assignment t' so the first clause is satisfied. All of the middle clauses contain some variable from the set $\{y_j^2, y_j^3, ...y_j^{k-3}\}$ which are all mapped to T by t'. Therefore, since each middle clause contains one of those variables the middle clauses are satisfied. The last clause contains z_{k-1} and z_k which one of these is said to be mapped to T by t, so the last clause must be satisfied. Thus, t' is a satisfying truth assignment if $l = k-1$ or k.

Scenario 3:

If $3 \leq l \leq k-2$, then set $t'(y_j^i) = T$ for $1 \leq i \leq l-2$ and $t'(y_j^i) = F$ for $l-1 \leq i \leq k-3$. So this means that some literal in $\{z_3, z_4, ...z_{k-2}\}$ must be mapped to T by t. It also tells us that $t'(\bar{y}_j^i) = F$ for $1 \leq i \leq l-2$ and $t(\bar{y}_j^i) = T$ for $l-1 \leq i \leq k-3$. For the first clause in C'_j both z_1 and z_2 are mapped to F by t, so $t'(y_j^1)$ must be true and it is by t'. Since $t'(\bar{y_j^{k-3}}) = T$ by the definition of t', the last clause is satisfied. In the middle clauses, we know that exactly one clause contains the literal z_l that is mapped to true by t. Thus, exactly one of the middle clauses is satisfied by t and all others must be satisfied by t'. By the definition of t' and the construction of our middle clauses to include variables $\bar{y}_j^i$ and y_j^{i+1} we are guaranteed by t' that one of these variables will be mapped to true, except in the clause containing z_l. The clause containing z_l is satisfied by t since $t(z_l) = T$. Therefore, t' is a satisfying truth assignment for $3 \leq l \leq k-2$.

So for any of the scenarios that fall under case 4, we see that t' is a satisfying truth assignment for C'_j. Since all of these clauses in C'_j are satisfied by t' and C' is the union of all C'_j's then C' must be satisfied by t'. The last thing needed to complete this transformation is to show that it can be completed in polynomial time. We know that our original instance of SAT was made up of a set of variables $U = \{u_1, u_2, ..., u_n\}$ and a set of clauses $C = \{c_1, c_2, ..., c_m\}$. Let the length of the input clauses = $|c_1| + |c_2| + ... + |c_m|$. By our setup of 3SAT above we know that if the original clause contained one literal $c_i = \{z_1\}$ we transformed it into $\{\{z_1, y_j^1, y_j^2\}, \{z_1, y_j^1, \bar{y}_j^2\}, \{z_1, \bar{y}_j^1, y_j^2\}, \{z_1, \bar{y}_j^1, \bar{y}_j^2\}\}$ which means if $|c_i| = 1$ then $|c'_i| = 12$. It follows that when literal $c_i = \{z_1, z_2\}$ we transformed it into $\{\{z_1, z_2, y_j^1\}, \{z_1, z_2, \bar{y}_j^1\}\}$. So if $|c_i| = 2$ then $|c'_i| = 6$. For the case when c_i contained three literals, c'_i was the exact same clause so if $|c_i| = 3$ then $|c'_i| = 3$. The last case was when the number of literals in c_i was greater than 3. When this was the case, we transformed c_i, where c_i contained k literals, into $k-2$ clauses in C' where each clause contains three literals. So if $|c_i| = k$ then $|c'_i| = 3(k-2) = 3|c_i| - 6$.

Suppose C is divided into two sets of clauses, where $c_1, c_2, ..., c_l$ are clauses in C with

3 or less literals and $c_{l+1}, c_{l+2}, ..., c_m$ are clauses with more than three literals. Then for clauses $c_1, c_2, ..., c_l$,

$$\{|c_1'| + |c_2'| + ... + |c_l'|\} \leq \underbrace{12 + 12 + ... + 12}_{\text{l times}} = 12l$$

and for clauses $c_{l+1}, c_{l+2}, ..., c_m$,

$$\underbrace{|c_{l+1}'| + ... + |c_m'|}_{m-(l-1)+1=m-l \text{ terms}} = (3|c_{l+1}| - 6) + ... + (3|c_m| - 6).$$

Now let $\gamma = \max\{|c_{l+1}|, ..., |c_m|\}$. By assigning γ equal to the size of the largest clause with more than three literals, we see that

$$[|c_1'| + |c_2'| + ... + |c_l'|] + [|c_{l+1}'| + ... + |c_m'|] \leq [12l] + [(m-l)(3\gamma - 6)].$$

Now given that the number of literals in our clause must be greater than or equal to l, $m \geq l$, we see that

$$12l + (m-l)(3\gamma - 6) \leq 12m + m(3\gamma - 6)$$

Note that $m \leq |c_1| + ...|c_m|$ and $\gamma \leq |c_1| + ...|c_m|$. Well we can define, $|c_1| + ... + |c_m|$ as our input length, I. So both m and γ are less than or equal to I. It follows that

$$12m + m(3\gamma - 6) \leq 12I + I(3I - 6).$$

This last part shows that our transformation function is less than or equal to the polynomial $3I^2 + 6I$ where I is the input length of our instance from SAT. This proves that the transformation is a polynomial transformation and thus 3SAT is NP-complete.

Let's look at an example to solidify the reader's understanding of this transformation. Let the following collection of variables and clauses:

$$U = \{u_1, u_2, u_3, u_4, u_5\} \text{ and } C = \{\{u_1, u_2\}, \{\bar{u}_3\}, \{u_2, u_4, \bar{u}_5\}, \{u_1, \bar{u}_2, u_3, u_4, \bar{u}_5\}\}$$

represent an instance of SAT. Let $t : U \to \{T, F\}$ be a satisfying truth assignment where $t(u_1) = T$, $t(u_2) = F$, $t(u_3) = F$, $t(u_4) = T$ and $t(u_5) = T$. It is easy to see that this truth function satisfies the given instance of SAT. Now we just need to transform a collection of variables and clauses, U and C representing a given instance of SAT into a collection of variables and clauses, U' and C', representing an instance of 3SAT. We do this by transforming each individual clause in SAT to a clause of set of clauses in 3SAT. We also create a truth function t', that is an extension of t, that is defined to be the same as t for variables in U but remains to be defined for the new variables we will create in $U' \setminus U$.

We begin with the first clause $\{u_1, u_2\}$. Notice that the $|\{u_1, u_2\}| = 2$, so we need to create an extra variable to include in the new clause in 3SAT. So we will add variable y_j^1, to the original clause to make it have three variables in it. So we transform clause $\{u_1, u_2\} \in C$ into two new clauses $\{u_1, u_2, y_1^1\}, \{u_1, u_2, \bar{y}_1^1\} \in C'$ of size three. Notice that since the original clause was satisfied by t then each of the new clauses will also be satisfied. So the first clause has been transformed.

Clause $\{\bar{u}_3\}$ is the second clause in C. Since this clause has one literal in it we will need to add two variables y_2^1 and y_2^2 from U' to make the new 3SAT clauses. We transform $\{\bar{u}_3\}$ into the four clauses $\{baru_3, y_2^1, y_2^2\}$, $\{baru_3, \bar{y}_2^1, y_2^2\}$, $\{baru_3, y_2^1, \bar{y}_2^2\}$ and $\{baru_3, \bar{y}_2^1, \bar{y}_2^2\}$. Since all of the new clauses in C' created from the $\{baru_3\}$ contain $\bar{u}_3$, and $t(\bar{u}_3) = T$, we know that all of the new clauses will be satisfied by t as well. So the second clause has been transformed.

The third clause, $\{u_2, u_4, \bar{u}_5\}$, contains three variables so we do not need to add any new variables to our clause. The new clause in C' will be identical to the original clause. So $\{u_2, u_4, \bar{u}_5\} \in C'$ and since $\{u_2, u_4, \bar{u}_5\}$ is satisfied by t the new clause is satisfied as well. So the third clause has been transformed.

SAT Instance	
Variables (U)	$u_1, u_2, u_3, u_4, u_5, \bar{u}_1, \bar{u}_2, \bar{u}_3, \bar{u}_4, \bar{u}_5$
Clauses (C)	$\{\{u_1, u_2\}, \{\bar{u}_3\}, \{u_2, u_4, \bar{u}_5\}, \{u_1, \bar{u}_2, u_3, u_4, \bar{u}_5\}\}$

Figure 4.1: Summary of SAT Instance

3SAT Instance	
Variables (U')	$u_1, u_2, u_3, u_4, u_5, \bar{u}_1, \bar{u}_2, \bar{u}_3, \bar{u}_4, \bar{u}_5, y_1^1, \bar{y}_1^1, y_2^1, \bar{y}_2^1$ $y_2^2, \bar{y}_2^2, y_4^1, \bar{y}_4^1, y_4^2, \bar{y}_4^2$
Clauses (C')	$\{u_1, u_2, y_1^1\}, \{u_1, u_2, \bar{y}_1^1\}, \{\bar{u}_3, y_2^1, y_2^2\}, \{\bar{u}_3, \bar{y}_2^1, y_2^2\}$ $\{\bar{u}_3, y_2^1, \bar{y}_2^2\}, \{\bar{u}_3, \bar{y}_2^1, \bar{y}_2^2\}, \{u_2, u_4, \bar{u}_5\}, \{u_1, \bar{u}_2, y_4^1\}$ $\{\bar{y}_4^1, u_3, y_4^2\}, \{\bar{y}_4^2, u_4, \bar{u}_5\}$

Figure 4.2: Summary of 3SAT Instance

The last clause $\{u_1, \bar{u}_2, u_3, u_4, \bar{u}_5\}$ has five literals in it. Since $|c_4| > 3$, we will need to create multiple clauses in C' to represent the original SAT clause. By following the transformation presented in this chapter, we see that we will need to create two new literals y_4^1 and y_4^2 in U' since $|c_4| = 5$. We then create three new clauses in C'. They are $\{u_1, \bar{u}_2, y_4^1\}$, $\{\bar{y}_4^1, u_3, y_4^2\}$ and $\{\bar{y}_4^2, u_4, \bar{u}_5\}$. Now set $t'(y_4^i) = F$ for $1 \leq i \leq 2$. So $\{u_1, \bar{u}_2, y_4^1\}$ is satisfied since $t'(u_1) = T$, $\{\bar{y}_4^1, u_3, y_4^2\}$ is satisfied since $t'(\bar{y}_4^1) = T$, and $\{\bar{y}_4^2, u_4, \bar{u}_5\}$ is satisfied since $t'(u_4) = T$. So our new clauses in C' are satisfied by t'.

So we have taken an instance of SAT that is satisfied, and transformed it into an instance of 3SAT that is satisfied.

3SAT to INDEPENDENT SET

In this chapter, we plan to show that the INDEPENDENT SET decision problem is in the set of NP-complete decision problems. We will do this by first showing that the INDEPENDENT SET problem is in NP. Once we have done that we will construct a polynomial transformation from Π_{3SAT} to Π_{IS}. Lastly, we will need to show that some set of clauses C making up an instance of 3SAT is satisfiable if and only if we can create a graph $G = (V,E)$, from the literals and clauses in that instance of 3SAT, and a positive integer $K \leq |V|$, such that G has a independent set of vertices of size K or greater. Once these steps have been completed, we will have shown that the INDEPENDENT SET problem is NP-complete.

The independent set problem, IS, is in the NP class of problems since, given a guessed set of vertices S', a nondeterministic turing machine can check whether or not the set is independent and if $|S'| \geq K$ in polynomial time.

Let $U = \{u_1, u_2, ..., u_n\}$ be a set of variables and $C = \{c_1, c_2, ..., c_m\}$ be a set of clauses making up any instance of 3SAT. Let $\bar{U} = \{\bar{u}_1, \bar{u}_2, ...\bar{u}_n\}$ be the set of negated variables from U. Each $c_i \in C$ consists of three literals and takes the form of $\{x_i, y_i, z_i\}$. We have to construct a graph $G = (V,E)$ and a positive integer $K \leq |V|$ such that G has an independent set of vertices of size K or larger if and only if C is satisfiable.

To create our graph $G = (V,E)$, we start by letting the variables in U and $\bar{U}$ be vertices in G. We then create an edge $\{u_i, \bar{u}_i\}$ for each variable $u_i \in U$. Note that since there are n variables in U there will be n edges in our graph of this type. For

each $c_i \in C$, we create a set of vertices $\{a_1[i], a_2[i], a_3[i]\}$, where $a_1[i]$ represents the first literal in clause i, $a_2[i]$ represents the second literal and $a_3[i]$ represents the third literal. For each set of vertices, $\{a_1[i], a_2[i], a_3[i]\}$, we will create the following edges: $\{(a_1[i], a_2[i]), (a_1[i], a_3[i]), a_2[i], a_3[i])\}$. The reader can visualize each clause $c_i \in C$ as an independent triangle in our graph G. To get a better picture of this, please see Figure **??**. Since there are m clauses in C we will have m triangles in our graph. To finish our graph we need to connect literals with the clauses in which they are used in C. We do this by connecting each vertex in the triangles with the vertex representing that literal. Since a clause in 3SAT contains exactly three variables, let each clause $c_j \in C$ be represented by the set of literals: $\{x_j, y_j, z_j\}$. For each $c_j \in C$ we create the following edges: $\{\{a_1[j], x_j\}, \{a_2[j], y_j\}, \{a_3[j], z_j\}\}$. These edges are referred to as "communication edges" in [**?**, p.54] where they discuss the polynomial transformation from 3SAT to VERTEX COVER. For example, if $c_1 = \{u_1, \bar{u}_3, \bar{u}_4\}$ then we would assign the following edges $\{\{a_1[1], u_1\}, \{a_2[1], \bar{u}_3\}, \{a_3[1], \bar{u}_4\}\}$ connecting the vertices from the triangle representing c_1 to it's negated literals. Lastly, let the integer K be defined so that it is equal to the sum of n and m. It should be noted that at most one vertex from each edge $\{u_i, \bar{u}_i\}$ and at most one vertex from each triangle $\{a_1[i], a_2[i], a_3[i]\}$ is in any independent set V'. So, for any independent set V', $|V'| \leq n + m = K$.

To get a better idea of what our graph G would look like consider the following example. Let an instance of 3SAT consist of a set of variables $U = \{u_1, u_2, u_3, u_4\}$ and a set of clauses $C = \{\{u_1, \bar{u_3}, \bar{u_4}\}, \{\bar{u_1}, u_2, \bar{u_4}\}\}$. In Figure **??**, you will see four edges at the top of our graph connecting each variable and it's negation. At the bottom of the graph you will see two triangles representing the two clauses from C. The top vertices and bottom triangle vertices are connected through edges that go from a literal in a clause (bottom vertex) to it's matching variable (top vertex). Since u_1 is the first literal in c_1, vertex $a_1[1]$ is connected to u_1. It follows that the other "communication edges" are

$\{\{a_2[1],\bar{u}_3\},\{a_3[1],\bar{u}_4\},\{a_1[2],\bar{u}_1\},\{a_2[2],u_2\},\{a_3[2],\bar{u}_4\}\}$. This example is adapted from an example for the VERTEX COVER decision problem in [**?**].

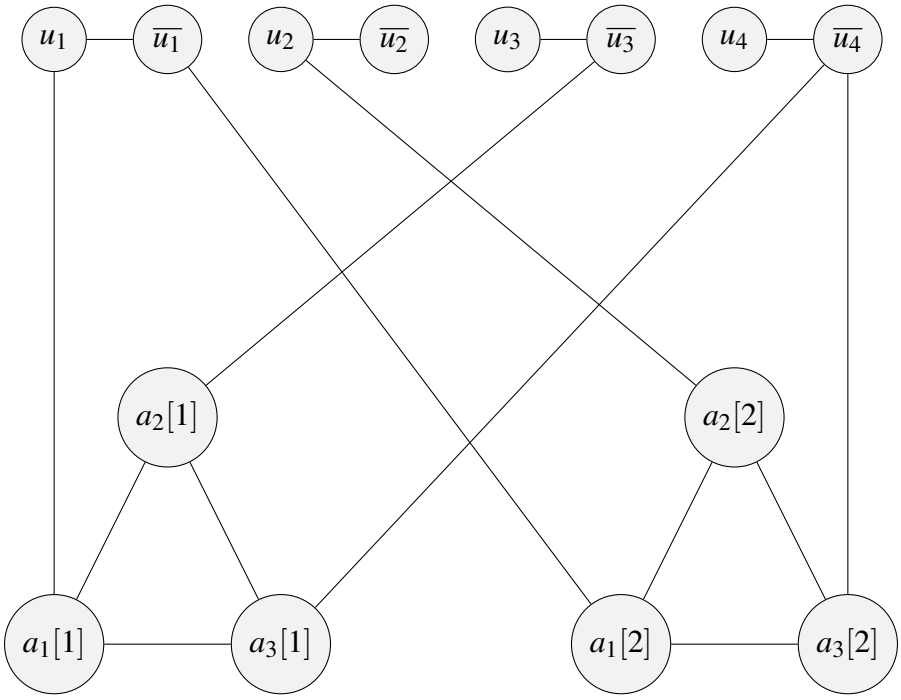

Figure 5.1: **INDEPENDENT SET graph based on a 3SAT instance. This example is described in the text.**

Now that we have mapped out a transformation from any instance of 3SAT to a graph $G = (V,E)$ representing an instance of IS we need to show that the set of clauses C making up an instance in 3SAT is satisfiable if and only if G has an independent set of vertices, V', with $|V'| \geq K$. We begin by showing that if G has an independent set of vertices, V' where $|V'| \geq K$ then C is satisfiable.

Assume that $V' \subseteq V$ is an independent vertex set in G with $|V'| \geq K$. Then by our graph setup $|V'| = K = n+m$, which means V' has exactly one vertex from each edge $\{u_i,\bar{u}_i\}$ and exactly one from each triangle $\{a_1[i],a_2[i],a_3[i]\}$. We need a truth assignment $t : U \to \{T,F\}$. Let $L = U \bigcup \bar{U}$. Given any truth assignment function $t : U \to \{T,F\}$, we

can extend $t : L \to \{T,F\}$ by setting for $u \in U$, $t(u) = T$ if and only if $t(\bar{u}) = F$. Likewise, $t(u) = F$ if and only if $t(\bar{u}) = T$.

Now we will define a truth function that maps a variable from L to either True or False depending on whether the vertex representing that variable is in V'. Define the truth assignment function $t : L \to \{T,F\}$ to be:

$$t(u) = F \text{ if } u \in V'$$
$$t(u) = T \text{ if } u \notin V'$$
$$t(\bar{u}) = F \text{ if } \bar{u} \in V'$$
$$t(\bar{u}) = T \text{ if } \bar{u} \notin V'$$

Let $c_i \in C$. Then $c_i = \{x_i, y_i, z_i\}$ where $x_i, y_i, z_i \in U$. By the definition of our "communication edges" in G, vertex x_i is adjacent to vertex $a_1[i]$ in triangle i. From the construction of G, we know that either vertex x_i or it's negation $\bar{x_i}$ is in the independent set V'. Also from the construction of G it follows that either y_i or $\bar{y_i}$ is in V', as well as either z_i or $\bar{z_i}$. We also know that exactly one of the vertices in each triangle $\{a_1[i], a_2[i], a_3[i]\}$ is in V'. Note that if $x_i \in V'$, by definition of our truth assignment function t, $t(x_i) = F$. The only way for clause c_i to not be satisfied is if $t(x_i) = F$, $t(y_i) = F$ and $t(z_i) = F$. This means that x_i, y_i, and z_i are in the independent set V'. Therefore, their negations $\bar{x_i}$, $\bar{y_i}$, and $\bar{z_i}$ are not in V'. Now notice that since x_i is in c_i, by the definition of the communication edges, there exists an edge from $a_1[i]$ to x_i. Similarly, there are communication edges from $a_2[i]$ to y_i and from $a_3[i]$ to z_i.

So, if x_i is in V' then $a_1[i]$ cannot be in V' since they are adjacent. Similarly, if y_i is in V' then $a_2[i]$ cannot be in V' and if z_i is in V' then $a_3[i]$ cannot be in V'. But one of $\{a_1[i], a_2[i], a_3[i]\}$ has to be in V', which is a contradiction. Therefore, c_i is satisfied. By similar reasoning, every clause in C must be satisfied.

So far we have shown that under our polynomial transformation that, if given a graph $G = (V,E)$ with an independent set of vertices V' such that $|V'| \geq K$, we have an instance

in 3SAT that is satisfiable. We now need to show that given a satisfiable instance in 3SAT under our transformation we have a graph $G = (V, E)$ with an independent set of vertices V' such that $|V'| \geq K$ to complete our polynomial transformation from Π_{3SAT} to Π_{IS}.

We begin by assuming that the collection of m clauses, C, making up some instance of 3SAT is satisfiable. Let $U = \{u_1, u_2, ..., u_n\}$ be a set of variables and $\bar{U} = \{\bar{u}_1, \bar{u}_2, ..., \bar{u}_n\}$ be the negated variables from U. Let $L = U \bigcup U'$. We know that since C is satisfiable, there exists a truth assignment $t : L \rightarrow \{T, F\}$ that makes each $c_i \in C$ satisfied. So, at least one literal in each c_i is mapped to true by t. Let $u \in V'$ if $u \in L$ and $t(u) = T$. So a vertex representing a literal in L is in the Independent Set of vertices if the truth assignment function t maps it to True.

Now let $c_j = \{x_j, y_j, z_j\}$ be the j^{th} clause in C. Since t was assumed to be a satisfying truth assignment for C, either $t(x_j) = T$, or $t(y_j) = T$, or $t(z_j) = T$. For each clause in C, consider the following scenarios to determine which vertex of $a_1[j]$, $a_2[j]$, $a_3[j]$ is selected to be in V':

Scenario 1: If $t(x_j) = T$, let $a_1[j] \in V'$. Note that $a_1[j]$ can only be adjacent to either x_j or $\bar{x}_j$ on top and $t(x_j) = T$ means $x_j \notin V'$ and $\bar{x}_j \in V'$. Since $\{a_1[j], x_j\}$ is an edge, $a_1[j]$ is not adjacent to any element of V' from the top vertices or the bottom vertices.

Scenario 2: If $t(x_j) = F$ and $t(y_j) = T$, let $a_2[j] \in V'$. Note that $a_2[j]$ can only be adjacent to either y_j or $\bar{y}_j$ on top and $t(y_j) = T$ means $y_j \notin V'$ and $\bar{y}_j \in V'$. Since $\{a_2[j], y_j\}$ is an edge, $a_2[j]$ is not adjacent to any element of V' from the top vertices or the bottom vertices.

Scenario 3: If $t(x_j) = F$ and $t(y_j) = F$, and $t(z_j) = T$ let $a_3[j] \in V'$. Note that $a_3[j]$ can only be adjacent to either z_j or $\bar{z}_j$ on top and $t(z_j) = T$ means $z_j \notin V'$ and $\bar{z}_j \in V'$. Since $\{a_3[j], z_j\}$ is an edge, $a_3[j]$ is not adjacent to any element of V' from the top vertices or the bottom vertices.

Clearly V' contains exactly one vertex from each edge $\{u_i, \bar{u}_i\}$ that connects a literal to

it's negation. V' also contains exactly one vertex from each triangle $\{a_1[i], a_2[i], a_3[i]\}$. By the construction of our graph each of the vertices from the edges $\{u_i, \bar{u}_i\}$ are independent and exactly one of these was selected to be in V'. The selected vertices from the triangles are also independent since there are no edges between any two triangles. The only thing that remains to be shown is that there is no edge between a vertex selected to be in V' from the edges $\{u_i, \bar{u}_i\}$ and a vertex selected from the triangles.

By the selection process described above, we see that exactly one vertex from each $\{a_1[j], a_2[j], a_3[j]\}$ in C is selected to be in V'. Note that there were m clauses in C. This gives us n vertices from each of the independent edges $\{u_i, \bar{u}_i\}$ and m vertices, one from each independent triangle representing the m clauses. Thus, we have a independent set of vertices V' with $m+n$ vertices. Since $m+n$ was defined to equal K, $|V'| = K$. Therefore $\Pi_{3SAT} \propto \Pi_{IS}$.

To end, let's refer back to the example that goes with Figure **??**. In that example, we had an instance of 3SAT that consisted of variables $U = \{u_1, u_2, u_3, u_4\}$, $\bar{U} = \{\bar{u}_1, \bar{u}_2, \bar{u}_3, \bar{u}_4\}$ and a set of clauses $C = \{\{u_1, \bar{u}_3, \bar{u}_4\}, \{\bar{u}_1, u_2, \bar{u}_4\}\}$. So we will create a graph $G = (V, E)$ from this instance of 3SAT. Let, $V = \{u_1, u_2, u_3, u_4, \bar{u}_1, \bar{u}_2, \bar{u}_3, \bar{u}_4, a_1[1], a_2[1], a_3[1], a_1[2], a_2[2], a_3[2]\}$. Now we will create the following edges, $\{u_i, \bar{u}_i\}$, for each $u_i \in U$. So far, the edges in G are $\{\{u_1, \bar{u}_1\}, \{u_2, \bar{u}_2\}, \{u_3, \bar{u}_3\}, \{u_4, \bar{u}_4\}\}$. But we still need to create the edges for the vertices that represent clauses c_1 and c_2. For $c_1 = \{u_1, \bar{u}_3, \bar{u}_4\}$ represented by vertices $\{a_1[1], a_2[1], a_3[1]\}$ we create edges $\{a_1[1], a_2[1]\}, \{a_1[1], a_3[1]\}, \{a_2[1], a_3[1]\}$. For $c_2 = \{\bar{u}_1, u_2, \bar{u}_4\}\}$ represented by vertices $\{a_1[2], a_2[2], a_3[2]\}$ we create the following edges: $\{a_1[2], a_2[2]\}, \{a_1[2], a_3[2]\}, \{a_2[2], a_3[2]\}$.

At this point our graph is almost complete. All of the vertices have been defined, most of the edges have been defined, the last thing to do is to construct the communication edges. Since the $c_1 = \{u_1, \bar{u}_3, \bar{u}_4\}$ we will construct the following communication edges: $\{a_1[1], u_1\}$, $\{a_2[1], \bar{u}_3\}$ and $\{a_3[1], \bar{u}_4\}$. Also we will need to construct three communication

edges for c_2. Those edges are $\{a_1[2], \bar{u}_1\}$, $\{a_2[2], u_2\}$ and $\{a_3[2], \bar{u}_4\}$. This now gives us the graph in Figure **??**.

Let $t : U \rightarrow \{T, F\}$ be a truth assignment function such that $t(u_1) = T$, $t(u_2) = T$, $t(u_3) = T$ and $t(u_4) = T$. Clearly, t satisfies both clauses in C. Now if $t(u) = T$ let $u \notin V'$ and if $t(u) = F$, let $u \in V'$. Similarly, if $t(\bar{u}) = T$ let $\bar{u} \notin V'$ and if $t(\bar{u}) = F$ let $\bar{u} \in V'$. So, u_1,u_2,u_3 and u_4 are not in V'. But their negations, $\bar{u}_1$,$\bar{u}_2$,$\bar{u}_3$ and $\bar{u}_4$ are in V'.

Now let's look at triangles that were created from the two clauses. Our first clause, $\{u_1, \bar{u}_3, \bar{u}_4\}$, was represented graphically by vertices $\{a_1[1], a_2[1], a_3[1]\}$. From the vertex selection process described in this chapter, we select $a_1[1]$ to be in V' since $t(u_1) = T$. Our second clause, $\{\bar{u}_1, u_2, \bar{u}_4\}$, was represented graphically by vertices $\{a_1[2], a_2[2], a_3[2]\}$. From the same selection process, we choose vertex $a_2[2]$ to be in V' since $t(\bar{x}_1) = F$ and $t(u_2) = T$. Now, our independent set of vertices V' includes $\bar{u}_1$, $\bar{u}_2$, $\bar{u}_3$, $\bar{u}_4$, $a_1[1]$ and $a_2[2]$. In the case of this example, there are two clauses, so $m = 2$ and four variables, so $n = 4$. Here our positive interger $K = m + n = 2 + 4 = 6$. So we have taken an instance of 3SAT and changed it into an instance of IS and shown that the graph has an independent set of vertices, V', such that $|V'| = 6$.

Bibliography

Bibliography

[1] Cook, Stephen A., "The complexity of theorem-proving procedures". in: Proceedings of the third annual ACM symposium on Theory of computing, 1971: ACM, New York, NY, USA, 151-158.

[2] Garey, Michael R. and Johnson, David S., *Computers and Intractability*, 1979: W.H Freeman and Company, New York.

[3] Karp, Richard M., "Reducibility Among Combinatorial Problems". Complexity of Computer Computations, 1972: Plenum Press, 85-103.

[4] Marcus, Daniel A., *Graph Theory - A Problem Oriented Approach*, 2008: The Mathematical Association of America, United States.

[5] Robson, J.M., "Algorithms for Maximum Independent Sets", Journal of Algorithms 7:No.2 (1986):425-440.

[6] Tarjan, Robert E. and Trojanowski, Anthony E., "Finding A Maximum Independent Set", SIAM Journal on Computing 6 (1977):537-546.

[7] Turing, Alan M., "On Computable Numbers", Proceedings of the London Mathematical Society 2:No.42 (1937):230-265.

[8] Wilf, Herbert S., *Algorithms and Complexity*, Second Edition, 2002: A K Peters, Ltd. Canada.

Vita

Andrew Schuyler Bristow IV was born in Richmond, Virginia on August 23rd, 1979. He graduated from Varina High School, Richmond, Virginia in 1997. After graduating high school he moved to Blacksburg, Virginia where he attended Virginia Polytechnic Institute and State University. At Virginia Tech, he earned his Bachelor of Science degree in Economics in 2002. Upon graduation he moved back to Richmond, Virginia and worked for both SunTrust Bank and the YMCA of Greater Richmond. In 2007, he began teaching mathematics at Patrick Henry High School in Ashland, Virginia. In December 2011 he will graduate from Virginia Commonwealth University, with a Master of Science degree in Mathematics. He currently resides with his wife, Amy, and their dog, George, in Richmond, Virginia.