Polynomially Reducing the Clique Problem to the SAT Problem with DIMACS Encoding

Austin J. Hunt

December 2, 2018

Introduction

The Clique Problem

The Clique Problem is an **NP-complete**[1] search problem in graph theory which centers on answering the question, ”Does a clique of size k exist in a given graph G = (V,E)?” where a **clique** is defined as a subset V’ of size k belonging to V in which every pair of vertices is adjacent. In other words, a clique of size k is a **complete subgraph** of G containing k vertices.

The Satisfiability Problem

Among the many other NP-complete problems that are studied by mathematicians and computer scientists today is the **Boolean Satisfiability Problem (SAT)**. With this problem, we ask, ”Does there exist an interpretation of Boolean formula B that satisfies it)?” That is, is it possible to assign specific True/False values to the Boolean literals in the formula B (ex: $x_1 \vee x_2 \wedge x_3$) such that B in its entirety becomes true? In the 1960s, this question sparked its own wave of algorithm development and theoretical research. In Stephen Cook’s 1971 paper entitled ”The Complexity of Theorem Proving Procedures,”, this problem became the first recognized NP-Complete problem via the well-known **Cook’s Theorem**. The next year, Richard Karp published ”Reducibility Among Combinatorial Problems” in which he used Cook’s theorem to show that 21 of the then-established computational problems were **polynomially reducible**[2] to the SAT problem, stamping them all as NP-complete. In the ’90s, the implications of Karp’s work culminated in a new class of software: *SAT solvers.*

SAT Solvers

As discussed by Victor W. Marek in his book (found at Addlestone Library) Introduction to Mathematics of Satisfiability, SAT Solvers are meant to take advantage of the reducibility dis-

[1]A problem P is NP-complete if 1) it is in the set of all NP problems and 2) all problems in that set can be polynomially reduced to P. While verifying a candidate solution for P can be done quickly, no algorithm exists that can *solve* **any instance** of P in polynomial time (though it may solve some small instances in polynomial time). As the size of P grows (where ”size” *here* refers to the number of vertices in the graph), the time required to solve P using any currently known algorithm increases exponentially.

[2]Reducible in polynomial time; if A is polynomially reducible to B, then all ”yes” instances of A are mapped one-to-one to ”yes” instances of B, and vice versa; this mapping can be established with a polynomial-time algorithm.

cussed by Karp; they were built to find solutions to NP-Complete search problems (like the Clique Problem) by first *encoding* those problems as propositional theories, or *boolean formulas.* Given an NP-complete search problem P, one translates P to a propositional theory T_p such that there exists a **one-to-one correspondence** between solutions to P and valuations that satisfy T_p. While these solvers certainly aren't the silver polynomial-time bullet for NP-complete problems (i.e. $P \overset{?}{=} NP$ is still unanswered), they have most definitely been applied in many areas: combinatorial optimization, electronic design automation, hardware and software verification, and much more.

The Goal

My goal is to define and implement a process of reducing instances of the Clique Problem to instances of the SAT Problem, and to then pass those reduced SAT instances to a pre-designed SAT solver. I want to observe the time required for these solvers to provide a "Yes" or "No" answer to the question, and to compare this time to the time required by a brute-force **exhaustive search algorithm**[3] for the Clique Problem that I will implement. I searched for and found Victor W. Marek's book An Introduction to Mathematics of Satisfiability in the Addlestone Library, and it has some super powerful, generalized insight on the process of encoding problems to DIMACS format[4], which I will use heavily in my own implementation. I also found another book titled Computers and Intractability: A Guide to the Theory of NP-Completeness, written by Michael Garey and David Johnson back in 1979 which offered some valuable information about SAT encoding. In short, I want to:

1. Encode any instance of the clique problem to DIMACS format
2. Pass that DIMACS encoded string into a pre-designed SAT solver
3. Let that SAT solver determine if that DIMACS formula is satisfiable or not
4. Interpret the output valuation (if it's satisfiable, the solver will give us values for the literals) or the "not satisfiable" result in the context of the Clique problem (valuation $\implies$ clique of size k exists consisting of vertices represented in some way by the Boolean literal values, "not satisifable" = "clique of size k does not exist"
5. Record the time taken for 1) Encoding the problem into DIMACS, 2) Solving the SAT problem, and 3) Decoding the result

DIMACS: The Center for Discrete Mathematics and Theoretical Computer Science

I will be using a SAT solver that accepts input in DIMACS format. This format is an encoded version of Conjunctive Normal Form, which consists of 1) any number of clauses joined by the

[3]For an input graph G = (V,E) and an integer k $<|\ V\ |$, exhaustively check all subgraphs of G of size k, and exhaustively search each pair of vertices within those subgraphs to determine if the subgraph is complete; store the subgraph as a clique if so.

[4]This is the commonly accepted input format for SAT solvers. It is a digitized form of CNF (conjunctive normal form), which is a type of Boolean formula consisting of a set of clauses connected by AND operators, each of which contain literals connected by or operators.

Boolean operator AND, 2) with each of those clauses containing one or more literals joined by the Boolean operator OR, where 3) each of those literals is either a positive literal or a negative literal (i.e. preceded by the negation operator).

To encode a CNF formula into DIMACS format, I must do the following:

i. Determine how many unique literals there are (where a literal and its negation do NOT count as separate literals) $\implies L$

ii. Determine how many clauses there are $\implies C$

iii. Start with the line:

$$p \quad cnf \quad L \quad C$$

   a. p is a reserved symbol, representing "problem type"
   b. cnf is a reserved symbol, indicating that the following data needs to be interpreted as a set of clauses
   c. L and C are the integer values found in steps [i.] and [ii.]

iv. Also, any line starting with c is a comment

v. Let integers 1...L be mapped bijectively to the set of L literals.

vi. As an example, use 1 -2 0 to represent the clause $literal_1 \vee \neg literal_2$, where 1 is mapped to $literal_1$, 2 is mapped to $literal_2$, and $literal_1, literal_2 \in L$; end each line with a 0 to represent the end of the clause

vii. Add a new line in the above format for each clause.

viii. NOTE: since CNF format was pre-specified on line 1, the Boolean operators OR and AND do not have to be encoded!

Mapping a Clique Problem Instance to a CNF Formula

The authors of the books mentioned above offered some nice discussions about encoding NP-Complete problems to CNF. This is perhaps the hardest part of this undertaking - primarily because my objective requires that I take a problem belonging to Graph Theory and translate it (create a one-to-one map) to a (nontrivial) Boolean Logic formula. The following is an explanation of how I plan to do this.

Let our generalized Clique Problem be, "Does a clique (C) of size k exist in a graph $G = (V, E)$?" Assume we are considering a specific instance of this problem (a specific graph with a specific value of k). From this instance, we begin the translation to a propositional theory.

1. We can define a list of variables x_{iv} for every $1 \leq i \leq k$ and every $v \in V$. We can interpret each of these variables x_{iv} as "v is the i^{th} vertex in the clique. This is a convenient way to start the mapping since each of these variables x_{iv} must be either **true** or **false**. That is, doing this gives us a functional set of literals. Now, we need to define some constraints that align with the bounds of the problem:

a. We know that for every i (where $1 \leq i \leq k$), there exists an i^{th} vertex in C (where C is the clique). Put simply, if there is a clique of size k in a given graph G, then there must be exactly one vertex v in that clique for each i between 1 and k. We can write this more formally as

$$\forall\{i \mid 1 \leq i \leq k\} \exists x_{iv}$$

b. For every **non-edge** $(v, w) \in E$, v AND w cannot BOTH be in the clique (since there must exist an edge between any pair of vertices in $(v, w) \in C$). We can write this more formally as

$$\forall\{i, j \mid i \neq j\} \; \forall\{v, w \in V | (v, w) \notin E \wedge v \neq w\} \; \neg x_{iv} \vee \neg x_{jw}$$

c. For every i, j (where $j \neq i$), the i^{th} vertex is different from the j^{th} vertex. That is, we know that a vertex v cannot be both the i^{th} and the j^{th} vertex in the clique. It also means that two different vertices cannot both be the i^{th} vertex in the clique. We can write this as a two-part constraint for simplicity:

$$\forall[i, j \mid i \neq j] \; \forall[v \in V] \; \neg x_{iv} \vee \neg x_{jv}$$

$$\forall i \; \forall[v, w \in V \mid v \neq w] \; \neg x_{iv} \vee \neg x_{iw}$$

If I encode the above constraints for any input graph $G = (V, E)$, I'll have a set of clauses for the CNF formula (in DIMACS format, since each boolean variable is represented as an integer). If there is a solution (i.e. if this formula is *satisfiable*), then the SAT solver will show us which of these x_{iv} must be true (and which must be false), which we i.e. which vertices must be in the clique!

Encoding a Clique Problem in DIMACS Format

The following is pseudocode for the algorithm that I plan to use for encoding any input instance of the Clique problem into DIMACS format. My plan for this algorithm is based on the mapping process described above. I will be including the Python file with this paper on submission, which contains much more detailed documentation. The Python file also contains the much less impressive exhaustive search algorithm that I developed (likely not originally since this is a famous problem); I won't include that one in this paper.

Algorithm 1 Encode the clique problem for a given graph represented as an adjacency matrix and a given k as DIMACS string for SAT solver processing

```
procedure EncodeCliqueToDIMACS(adjMatrix,k)
    $numVertices \leftarrow len(adjMatrix)$
    $literal \leftarrow 1$
    $literal\_dict \leftarrow \{\}$
    $DIMACS \leftarrow$ ""
    $clause\_list \leftarrow []$
    for $i \leftarrow 1$ to $k$ do                                  ▷ Encode Constraint 1
        $clause \leftarrow$ ""
        for $v \leftarrow 0$ to $numVertices - 1$ do
            $literal\_dict[literal] \leftarrow [i, v]$
            $clause \leftarrow clause + str(literal)$
            $literal \leftarrow literal + 1$
        $clause \leftarrow clause +$ "0"
        $clause\_list.append(clause)$
    for $literal1$ in $literal\_dict$ do    ▷ Encode Constraint 2 AND Constraint 3; both require the same nested loop structure
        $i \leftarrow literal\_dict[literal1][0]$
        $v \leftarrow literal\_dict[literal1][1]$
        for $literal2$ in $literal\_dict$ do
            $j \leftarrow literal\_dict[literal2][0]$
            $w \leftarrow literal\_dict[literal2][1]$                ▷ Constraint 2 condition
            if $adjMatrix[v][w] = 0$ and $v \neq w$ and $literal1 \neq literal2$ then
                $clause \leftarrow$ "-" $+str(literal1)+$" -"$+str(literal2) +$ "0"
                if $clause$ not in $clause\_list$ then
                    $clause\_list.append(clause)$
                                                                  ▷ Constraint 3 condition
            if $(v = w$ and $i \neq j)$ or $(v \neq w$ and $i = j)$ and $literal1 \neq literal2$ then
                $clause \leftarrow$ "-" $+str(literal)+$ " -" $+str(literal2)+$ " 0"
                if $clause$ not in $clause\_list$ then
                    $clause\_list.append(clause)$
    $numLiterals \leftarrow len(literal\_dict)$
    $numClauses \leftarrow len(clause\_list)$
    $DIMACS \leftarrow$ ""
    $firstLine \leftarrow$ "p cnf " + $str(numLiterals)$ + " " + $str(numClauses)$
    $DIMACS \leftarrow DIMACS + frstLine$                              ▷ add clauses
    for $c$ in $clause\_list$ do
        $DIMACS \leftarrow DIMACS +$ "\n" $+c$
    return $DIMACS$
```

Solving and Interpreting Results: A Web Dev Approach

For testing purposes, I developed a small web application using **Flask** that enables me to avoid having to manually copy and paste all of the individual DIMACS strings into the online SAT solver. Instead, I first copied the minisat.js file to my local computer, and then I modified the SAT solver's inherent **solve** function so that I could loop through a list of generated DIMACS strings, pass each one in, and store their corresponding solutions output by the function. On top of that, I used a multiprocessing approach on a Windows VM through Microsoft Azure to generate all of the DIMACS strings for the different Clique Problem instances. It should be noted that I **did** include test cases for graphs with 1000 nodes and 2000 edges, but these DIMACS strings were taking a **very** long time to generate ($> 1 hour$), so I decided to limit my tests to graphs with 100 nodes and 200 edges at maximum. For each value of k in the list [2,6,11], I generated one random graph of each of the following dimensions:

a. 5 vertices, 4 edges

b. 10 vertices, 15 edges

c. 20 vertices, 30 edges

d. 50 vertices, 80 edges

e. 100 vertices, 200 edges

and I subsequently generated the DIMACS string for each problem instance. That is, for every k, I generated 3 Clique problem instances - one for each graph size - and I then generated the corresponding DIMACS string.

The following are the results obtained from passing the DIMACS strings into an online sat solver for the k values of 2, 5, and 11.

k	**| V |**	**| E |**	**SAT Solver Output & Time**	**Exhaustive Search Output & Time**
2	5	4	SAT 1 -2 -3 -4 -5 -6 -7 8 -9 -10, CPU time: 0.004s	[(0, 2), (0, 3), (1, 4), (2, 3)], CPU time: 0.002s
2	10	15	SAT -1 2 -3 -4 -5 -6 -7 ..., CPU time: 0.002s	[(0, 1), (0, 3), (0, 4),...] , CPU time: 0.0001s
2	20	30	SAT -1 2 -3 -4 -5 -6 -7 ..., CPU time: 0.011s	[(0, 1), (0, 3), (0, 4), (0, 8), (1, 2), (1, 3), (1, 5), ...], CPU time: 0.0016s
2	50	80	SAT -1 -2 -3 -4 -5 -6 -7 ..., CPU time: 0.044s	[(0, 27), (0, 28), (0, 33), (0, 44),...], CPU time: .0012s
2	100	200	SAT -1 -2 -3 -4 -5 -6 -7...,CPU time: 0.696s	[(0, 30), (0, 44), (0, 65), (0, 77),...], CPU time: .0417s
6	10	15	UNSAT, CPU time: 0.003s	False , CPU time: 0.007s
6	20	30	UNSAT, CPU time: 0.538s	False, 0.1406s
6	50	80	UNSAT, CPU time: 1.764s	False, CPU time: 51.561s
6	100	200	UNSAT, CPU time: 11.762s	False, CPU time: 3874.0706s
11	20	30	UNSAT, CPU time: 9.4s	False, CPU time: 0.5971s
11	50	80	UNSAT 1.511s (INTERESTING)	??? , CPU time: > 10 minutes
11	100	200	ERROR (DIMACS string too long)	???, > 1 hour

It can very easily be seen that while the exhaustive search approach to finding cliques actually works slightly faster than the SAT solver for very small problem sizes, the exhaustive search algorithm becomes very inefficient very quickly relative to the SAT solver as the problem size grows to 100 (where size refers to the number of vertices). It should be noted that when k exceeds the cardinality of V, the row is skipped (ex: no row for k = 6, $\mid V \mid= 5, \mid E \mid= 4$) since there cannot be a clique of size k > $\mid V \mid$ (a subset of set S cannot have larger cardinality than S itself).

Interpreting the SAT Solver Valuation

To interpret the SAT solver output (let us assume for a given instance I, I is satisfiable) in the context of the clique problem, we need to do two things:

1. Break the output into the list of literals; that is, split them on spaces, assuming the SAT solver outputs in the format

$$literal_1 \; literal_2 \; literal_3 \; .... \; literal_n$$

2. Loop through the list, gather all those which are not preceded by "-"

Since each literal (integer) corresponds to one specific $x_i v$ variable (refer to the previously used predicate logic in the mapping process), we know that the only vertices (v) in the clique are those corresponding to literals NOT preceded by "-". Thus, we can determine the vertices in our clique in linear time by looping through the list of literals and appending to a list (in constant time) the vertices corresponding to our positive literal. These are stored in the dictionary $literal_dict$ generated when creating the DIMACS string. Remember that the (key, value) pairs in our dictionary look as follows:

$$literal : [i, v]$$

indicating that this literal represents the statement, ”v is the ith vertex of the clique.” To give an example, let’s look at the first row of the above table. The SAT solver output was

$$SAT 1 - 2 - 3 - 4 - 5 - 6 - 78 - 9 - 10$$

. The problem instance was k = 2 for a graph of 5 vertices and 4 edges, with vertices numbered 0...4. So, going from 1 to 10, our literals represent the following statements:

1 $\implies$ v_0 is the first vertex of the clique

-2 $\implies$ v_1 is NOT the first vertex of the clique

-3 $\implies$ v_2 is NOT the first vertex of the clique

-4 $\implies$ v_3 is NOT the first vertex of the clique

-5 $\implies$ v_4 is NOT the first vertex of the clique

-6 $\implies$ v_0 is NOT the second vertex of the clique

-7 $\implies$ v_1 is NOT the second vertex of the clique

8 $\implies$ v_2 is the second vertex of the clique

-9 $\implies$ v_3 is NOT the second vertex of the clique

-10 $\implies$ v_4 is NOT the second vertex of the clique

So, the SAT solver tells us that our clique consists of vertices 0 and 2, which is consistent with the first clique in the list returned by the exhaustive search. NOTE THAT FOR K = 2, ANY PAIR OF ADJACENT VERTICES QUALIFIES AS A CLIQUE, WHICH IS WHY THERE ARE SO MANY CLIQUES FOR K = 2.

Idea for Expansion

For a little over a year now, I’ve been working as the sole developer for an online platform that teaches Python called PolyPy.com, so I have a very strong interest in web development (particularly using the Django framework). I think it would be a brilliant idea to build a platform that does the above encoding for not just the Clique Problem, but for many possible problems (both NP-hard and not NP-hard). A user could simply go to the website, specify the problem type, specify the exact instance of the problem, and click ”Generate DIMACS”. The backend would use the constraints for the input problem type, and it would generate the DIMACS string corresponding to the input problem instance. This does NOT exist, and I think something like this could be used extensively, perhaps even just for testing SAT solvers on a wide variety of inputs.

References

[1] Cook, Stephen A. The Complexity of Theorem-Proving Procedures. Proceedings of the Third Annual ACM Symposium on Theory of Computing - STOC '71, 1971, doi:10.1145/800157.805047.

[2] Garey, Michael R. and Johnson, David S. *Copmuters and Intractability [A guide to the Theory of NP-Completeness].* W. H. Freeman and Company, 1979. [*On the electrodynamics of moving bodies*]. Annalen der Physik, 322(10):891921, 1905.

[3] Karp, Richard M. Reducibility Among Combinatorial Problems. 50 Years of Integer Programming 1958-2008, 2009, pp. 219241., doi:10.1007/978-3-540-68279-0_8.

[4] Marek, V. Viktor. Introduction to Mathematics of Satisfiability. Chapman & Hall, 2009

[5] Sabharwal, Ashish. (2011). Modern SAT Solvers: Key Advances and Applications [Powerpoint]. Retrieved from IBM Watson Research Site.