CS1200: Intro. to Algorithms and their Limitations Anshu & Vadhan

Problem Set 7

Harvard SEAS - Fall 2024 *Due: Wed Nov. 13, 2024 (11:59pm)*

Your name: Ethan Veghte
Collaborators:
No. of late days used on previous psets: 8
No. of late days used after including this pset: 8

The purpose of this problem set is to develop skills in implementing graph algorithms, appreciate the impact of different kinds of worst-case exponential algorithms in practice, and practice reducing problems to SAT.

1. (Another coloring algorithm) In the Github repository for PS7, we have given you basic data structures for graphs (in adjacency list representation) and colorings, an implementation of the coloring algorithm from ps5, and a variety of test cases (graphs) for coloring algorithms. For Windows users, we will work on getting a Google Colab up and running, so check Ed for updates – in the meantime, we recommend trying to run your code in WSL.

   (a) Implement the reduction from 3-coloring to SAT given in class in the function `sat_3_coloring`, producing an input that can be fed into the SAT Solver Glucose, and verify its correctness by running `python3 -m ps7_tests 3`.

   (b) Compare the efficiency of Exhaustive-Search 3-coloring, the $O(1.45^n)$-time MaximalIS+2COL algorithm for 3-coloring from problem set 5 (feel free to use the staff solution or your own implementations from problem set 5), and your implementation from Part 1a using `ps7_experiments`. In the experiments file, we've provided code to generate two types of graphs (lines of rings and clusters of independent sets) and some new hard graph instances. For each of those types of graphs, how many of the given instances, if any, can each algorithm solve within 10 seconds (same time limit as problem set 5)? You should fill out the table and briefly discuss your findings, as well as why these algorithms perform the way they do given what we have discussed in lecture.

Algorithm	Exhaustive	ISET BFS	SAT Color
# Solvable Ring Instances	$n = 15$	$n = 1500$	$> 600,000$
# Solvable Cluster Instances	$n = 20$	90 or 80	$n = 5960$
# Solvable Hard Graphs	< 20	80 nodes	inconsistent

   (a) ***Solvable Ring Instances:*** *First, Exhaustive search algorithm was obviously ineffective. As expected it could not handle large complexity graphs and solve in* < 10 *seconds because with increase in complexity the number of colorings increases exponentially. The highest complexity ring instance was one with* $n = 15$*, and by* $n = 16$ *it timed out. BFS, worked decently well and could get up to a node count of 1500, which in comparison to Exhaustive search was much better. However, most impressively, SAT Color algorithm could solve an instance with extremely high node count in under 10 seconds. I did not find an upper bound but got as high as 600,000 nodes without it timing out. I did not go further as*

it would take multiple minutes to generate a graph of that complexity, for it only to be solved in < 2 *seconds.*

(b) ***Solvable Cluster Instances:*** *I was able to reduce the complexity down to one with 20 nodes and that was the largest complexity taht exhaustive search was able to complete. BFS worked decently well, but was dependent on the number of clusters. With 2 cluster graphs, it could solve up to 90 nodes, however if it were 3 or more it generally timed out around 80 or fewer nodes. The SAT Color algorithm was once again quite impressive as it completed a Cluster instance with 5860 nodes in under 10 seconds. That was the highest complexity success as it timed out at* $n = 6280$.

(c) ***Solvable Hard Graphs:*** *I could not find an upper bounded Hard Graph that exhaustive search could solve, because the hard graphs were specified under /hardinstances, and 20 was the lowest complexity one, and exhaustive search still timed-out. BFS got to 80 nodes consistently with solvable hard graphs, but would timeout afterwards. This is a respectable number, but in comparison to SAT Color algorithm it was nothing. SAT Color is super impressive with regards to solving Hard Graphs. It solved a 197912 node instance in 7 seconds with a 164970 node instance being solved in 5 seconds. However, I suspect any tangible further increase in the size of the instance would create the upper bound in what SAT Color algorithm can solve in a 10 second window. I also noticed, though, that SAT Color algorithm would also sometimes fail at a 500 node instance which would be confusing to me.*

Reasonings

(a) ***Exhaustive search:*** *This method was bound to struggle in solving these 3-coloring problems because it has to find all possible colorings which increases exponentially as the number of nodes increases. As such, even small graphs in cluster instances and ring instances failed. Furthermore, hard graphs generally add complexity to the graphs that make finding a successful coloring more difficult, but likely the configuration meant there were more possible 3-colorings even though there were few nodes. This is likely why it timed-out so early.*

(b) ***ISET BFS:*** *BFS relies on maximal independent sets, and can do pretty well with ring instances because they are generally sparse graphs. This is because finding a maximal independent set with sparse graphs to break down the graph into simpler parts and recursively apply 2-coloring, is effective on these simpler graphs. However, when you increase complexity, the efficiency declines significantly. For solvable cluster instances, it was better with 2 clusters reaching 90 nodes, but more clusters meant worse performance. This is because with more clusters it becomes harder to find maximal indpendent sets. Same is true for this method on hard graphs whose complex configuration generally tries to make these methods slow down. With breadth first search here, it becomes difficult to find maximal independent sets because the graphs are so interconnected that any vertex added is likely to have an edge with a vertex already added. This slows down the algorithm's ability to successfully find a maximal independent set.*

(c) ***SAT Color:*** *By encoding these graphs and problems as satisfiability constraints, the efficiency of the solvers increases significantly. This is because with boolean constraints the solver is able to efficiently eliminate possible colorings that cannot lead to a solution,*

this means it is able to vastly reduce the number of possibilities it needs to check. For organized graph structures like ring instances and cluster instances, the SAT solver can use the graphs internal structure to simplify the clauses, and remove the need to check for every possible coloring. This significantly surpasses the capabilities of the other two algorithms. For hard graphs, since there is likely less consistency as to the internal structure, which would allow the clauses to be reduced and simplified, SAT color is less consistent. Generally it was still extremely effective, although there were some instances where it would time out at 500 nodes, and others where it would reach 197,000.

2. (Resolution) Use the algorithm `ResolutionInOrder` that we saw in Lecture 16 to decide the satisfiability of the following formulas, and use the algorithm `ExtractAssignment` to obtain a satisfying assignment for any that are satisfiable. (Please make sure to follow both algorithms *exactly*, including the order in which the clauses are processed. A correct final solution that does not show all of the intermediate steps of both algorithms will not receive full score.)

   (a) $\varphi(x_0, x_1, x_2, x_3) = (x_0 \vee x_1) \wedge (\neg x_2 \vee x_1 \vee x_3) \wedge (x_3 \vee \neg x_1) \wedge (\neg x_3) \wedge (x_1 \vee x_2)$.

   First we understand the setup of the clauses to be
   $C_0 = (x_0 \vee x_1)$
   $C_1 = (\neg x_2 \vee x_1 \vee x_3)$
   $C_2 = (x_3 \vee \neg x_1)$
   $C_3 = (\neg x_3)$,
   $C_4 = (x_1 \vee x_2)$
   The first step is to simplify each clause, but since none of the clauses have any redundant repititions, nothing is necessary.
   Now we start finding the resolutions. Starting with C_0 *and* C_1.
   $C_0 \diamond C_1 = (x_0 \vee x_1) \diamond (\neg x_2 \vee x_1 \vee x_3)$
   In the algorithm if $R = C_i \diamond C_j$ *and if* $R = 0$ *then it is unsatisfiable. However we can see for* $R = C_1 \diamond C_0$ *that* $R \neq 0$ *then we continue algorithm. Because in this case there is no literal* $l \in C_0$ *and* $\neg l \in C_2$ *then there is no resolvent and it equals 1. This method will continue throughout this process.*
   I will now move faster through the next steps as that is one iteration.
   $C_0 \diamond C_2 = (x_0 \vee x_1) \diamond (x_3 \vee \neg x_1) = (x_0 \vee x_3)$
   $R = C_0 \diamond C_2 \notin \{0, 1\} \Rightarrow$
   $C_5 = (x_0 \vee x_3)$
   $C_0 \diamond C_3 = (x_0 \vee x_1) \diamond (\neg x_3) = 1$
   $R = 1$ *so move on*
   $C_0 \diamond C_4 = (x_0 \vee x_1) \diamond (x_1 \vee x_2) = 1$
   $R = 1$ *so move on*
   That is the end of the first i iteration. We know move on with $i = 1$ *and* $f = 6$.
   $C_1 \diamond C_2 = (x_1 \vee \neg x_2 \vee x_3) \diamond (x_3 \vee \neg x_1) = (\neg x_2 \vee x_3)$
   $R = C_1 \diamond C_2 \notin \{0, 1\} \Rightarrow$
   $C_6 = (\neg x_2 \vee x_3)$
   $C_1 \diamond C_3 = (x_1 \vee \neg x_2 \vee x_3) \diamond (\neg x_3) = (x_1 \vee \neg x_2)$
   $R = C_1 \diamond C_3 \notin \{0, 1\} \Rightarrow$
   $C_7 = (x_1 \vee \neg x_2)$

$C_1 \diamond C_4 = (x_1 \vee \neg x_2 \vee x_3) \diamond (x_1 \vee x_2) = (x_1 \vee x_3)$
$R = C_1 \diamond C_4 \notin \{0, 1\} \Rightarrow$
$C_8 = (x_1 \vee x_3)$
$C_1 \diamond C_5 = (\neg x_2 \vee x_1 \vee x_3) \diamond (x_0 \vee x_3) = 1$
$R = 1$ *so move on*
For sake of reference I want to put set of Cs together to make sure there are no conflicts (i put them in 3s so it is easier to see):
$(x_0 \vee x_1)_0, (\neg x_2 \vee x_1 \vee x_3)_1, (x_3 \vee \neg x_1)_2,$
$(\neg x_3)_3, (x_1 \vee x_2)_4, (x_0 \vee x_3)_5,$
$(\neg x_2 \vee x_3)_6, (x_1 \vee \neg x_2)_7, (x_1 \vee x_3)_8$
Now we move to C_2. $f = 9$, $i = 2$
$C_2 \diamond C_3 = (x_3 \vee \neg x_1) \diamond (\neg x_3) = (\neg x_1)$
$R = C_2 \diamond C_3 \neq 0 \Rightarrow$
$C_9 = (\neg x_1)$
$C_2 \diamond C_4 = (x_3 \vee \neg x_1) \diamond (x_1 \vee x_2) = (x_2 \vee x_3)$
$R = C_2 \diamond C_4 \neq 0 \Rightarrow$
$C_{10} = (x_2 \vee x_3)$
$C_2 \diamond C_5 = (x_3 \vee \neg x_1) \diamond (x_0 \vee x_3) = 1$
$R = 1$ *so move on*
$C_2 \diamond C_6 = (x_3 \vee \neg x_1) \diamond (\neg x_2 \vee x_3) = 1)$
$R = 1$ *so move on*
$C_2 \diamond C_7 = (x_3 \vee \neg x_1) \diamond (x_1 \vee \neg x_2) = (\neg x_2 \vee x_3)$
This exists already in clauses as C_6 *so move on*
$C_2 \diamond C_8 = (x_3 \vee \neg x_1) \diamond (x_1 \vee x_3) = (x_3)$
$R = C_2 \diamond C_8 \neq 0 \Rightarrow$
$C_{11} = (x_3)$
We now enter next loop for C_3, *with* $f = 12$ $i = 3$. $C_3 = \neg x_3$
We will enter into this loop, because $f > i + 1$ $(f = 12, i = 3)$. *I am going to check over the list of Cs to check whether it will finish this loop iteration. As we can see, though, since* $C_3 = (\neg x_3)$ *and* $C_{11} = (x_3)$ *when we get to* $C_3 \diamond C_{11}$ *we will get 0, which means that the algorithm is unresolvable. We made it to this loop and it will enter into it, however it will not successfully exit this loop because it will reach* $C_3 \diamond C_{11}$ *which will output 0, proving, as mentioned that it is unresolvable.*

(b) $\varphi(x_0, x_1, x_2, x_3) = (x_0 \vee x_1 \vee \neg x_3) \wedge (x_2 \vee x_3) \wedge (x_0 \vee \neg x_2) \wedge (x_2)$.

Same process as above. First set up initial C clauses:
$C_0 = (x_0 \vee x_1 \vee \neg x_3),$
$C_1 = (x_2 \vee x_3),$
$C_2 = (x_0 \vee \neg x_2),$
$C_3 = (x_2)$
Then start looping process to find resolutions of, first, C_0.
$C_0 \diamond C_1 = (x_0 \vee x_1 \vee \neg x_3) \diamond (x_2 \vee x_3) = (x_0 \vee x_1 \vee x_2)$
$R = C_0 \diamond C_1 \notin \{0, 1\} \Rightarrow$
$C_4 = (x_0 \vee x_1 \vee x_2)$

$C_0 \diamond C_2 = (x_0 \vee x_1 \vee \neg x_3) \diamond (x_0 \vee \neg x_2) = 1$
$R = 1$ *so move on*
Because in this case there is no literal $l \in C_0$ *and* $\neg l \in C_2$ *then there is no resolvent and it equals 1. This method will continue throughout this process.*
$C_0 \diamond C_3 = (x_0 \vee x_1 \vee \neg x_3) \diamond (x_2) = 1$
$R = 1$ *so move on*
Next loop iteration with C_1*, so* $f = 7$*,* $i = 1$ *:*
$C_1 \diamond C_2 = (x_2 \vee x_3) \diamond (x_0 \vee \neg x_2) = (x_0 \vee x_3)$
$R = C_1 \diamond C_2 \neq 0 \Rightarrow$
$C_5 = (x_0 \vee x_3)$
$C_1 \diamond C_3 = (x_2 \vee x_3) \diamond (x_2) = 1$
$R = 1$ *so move on*
$C_1 \diamond C_4 = (x_2 \vee x_3) \diamond (x_0 \vee x_1 \vee x_2) = 1$
$R = 1$ *so move on*
$C_1 \diamond C_5 = (x_2 \vee x_3) \diamond (x_0 \vee x_3) = 1$
$R = 1$ *so move on*
Next loop iteration with $i = 2$*,* $f = 6$
$C_2 \diamond C_3 = (x_0 \vee \neg x_2) \diamond (x_2) = (x_0)$
$R = C_2 \diamond C_3 \notin \{0, 1\} \Rightarrow$
$C_6 = (x_0)$
$C_2 \diamond C_4 = (x_0 \vee \neg x_2) \diamond (x_0 \vee x_1 \vee x_2) = (x_0 \vee x_1)$
$R = C_2 \diamond C_4 \not\in \{0, 1\} \Rightarrow$
$C_7 = (x_0 \vee x_1)$
$C_2 \diamond C_5 = (x_0 \vee \neg x_2) \diamond (x_0 \vee x_3) = 1$
$R = 1$ *so move on*
Next loop iteration with C_3 *so* $f = 8$*,* $i = 3$
$C_3 \diamond C_4 = (x_2) \diamond (x_0 \vee x_1 \vee x_2) = 1$
$R = 1$ *so move on*
$C_3 \diamond C_5 = (x_2) \diamond (x_0 \vee x_3) = 1$
$R = 1$ *so move on*
$C_3 \diamond C_6 = (x_2) \diamond (x_0) = 1$
$R = 1$ *so move on*
For sake of reference I want to put set of Cs together to make sure there are no conflicts (i put them in 3s so it is easier to see):
$(x_0 \vee x_1 \vee \neg x_3)_0, (x_2 \vee x_3)_1, (x_0 \vee \neg x_2)_2$
$(x_2)_3, (x_0 \vee x_1 \vee x_2)_4, (x_0 \vee x_3)_5$
$(x_0)_6$
Next loop iteration with C_4*, so* $i = 4$*, and* $f = 7$*.*
$C_4 \diamond C_5 = (x_0 \vee x_1 \vee x_2) \diamond (x_0 \vee x_3) = 1$
$R = 1$ *so move on*
$C_4 \diamond C_6 = (x_0 \vee x_1 \vee x_2) \diamond (x_0) = 1$
$R = 1$ *so move on*
Next iteration for C_5 *where* $i = 5$ *and* $f = 7$*.*
$C_5 \diamond C_6 = (x_0 \vee x_3) \diamond (x_0) = 1$
$R = 1$ *so move on*

Before entering into next loop we check whether $f > i+1$ (not explicitly done in previous iterations, but they passed so I did not write it). In this case $i = 6$ and $f = 7$, so it is not true that $7 > 6 + 1$. As a result the algorithm exits and returns satisfiable. We will now use `ExtractAssignment` *to get satisfying assignment.*
First lets remind ourselves of set C of clauses for ease of reference.
$C_0 = (x_0 \vee x_1 \vee \neg x_3)$,
$C_1 = (x_2 \vee x_3)$,
$C_2 = (x_0 \vee \neg x_2)$,
$C_3 = (x_2)$
$C_4 = (x_0 \vee x_1 \vee x_2)$
$C_5 = (x_0 \vee x_3)$
$C_6 = (x_0)$
From there we first check if there are singleton clauses, which we clearly see there to be 2 (C_3 and C_6). As such we set $x_0, x_2 = 1$.
Now we update the clauses accordingly. Since C_3, C_6 have been satisfied we can remove them entirely. Furthermore, since x_2 is true $C_1 = (x_2 \vee x_3)$ has also been satisfied and can be removed. The same goes for $C_0 = (x_0 \vee x_1 \vee \neg x_3)$, & $C_2 = (x_0 \vee \neg x_2)$, & $C_4 = (x_0 \vee x_1 \vee x_2)$, & $C_5 = (x_0 \vee x_3)$ because $x_0 = 1$ (i.e. true). As such since all of the clauses have been satisfied $\{C_0$ by $x_0 = 1$, C_1 by $x_2 = 1$, C_2 by $x_0 = 1$, C_3 by $x_2 = 1$, C_4 by $x_0 = 1$, C_5 by $x_0 = 1$, and C_6 by $x_0 = 1$, and we only now that $\varphi(x_0, x_1, x_2, x_3) = (1, x_1, 1, x_3)$ so far, then we can arbitrarily set x_1 and x_3. In other words, since all clauses are satisfied by $x_0 = 1$ and $x_2 = 1$ then x_1 and x_3 can be set arbitrarily. Lets say they are 0, and 1 respectively. In the end a satisfying assignment of $\varphi(x_0, x_1, x_2, x_3) = (1, 0, 1, 1)$

(c) $\varphi(x_0, x_1, x_2) = (x_0) \wedge (\neg x_0 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$.

Same process as above. First set up initial C clauses:
$(x_0), (\neg x_0 \vee x_1 \vee x_2), (\neg x_1 \vee \neg x_2)$
Now we start with $i = 0$ and $f = 3$, by doing resolutions of clauses with C_0.
$C_0 \diamond C_1 = (x_0) \diamond (\neg x_0 \vee x_1 \vee x_2) = (x_1 \vee x_2)$
$R = C_0 \diamond C_1 \neq 0 \Rightarrow$
$C_3 = (x_1 \vee x_2)$
$C_0 \diamond C_2 = (x_0) \diamond (\neg x_1 \vee \neg x_2) = 1$
$R = 1$ *so move on*
Next iteration with C_1, i =1, f = 4
$C_1 \diamond C_2 = (\neg x_0 \vee x_1 \vee x_2) \diamond (\neg x_1 \vee \neg x_2) = (\neg x_0)$
$R = C_1 \diamond C_2 \neq 0 \Rightarrow$
$C_5 = (\neg x_0)$
$C_1 \diamond C_3 = (\neg x_0 \vee x_1 \vee x_2) \diamond (x_1 \vee x_2) = 1$
$R = 1$ *so move on*
Next iteration with C_2, i = 2, f=4
$C_2 \diamond C_3 = (\neg x_1 \vee \neg x_2) \diamond (x_1 \vee x_2) = (\neg x_2 \vee x_2)$ *which is always true*
Moving on, we have reached a point where it is untrue that $f > i + 1$ because $f = 4$ and $i = 3$ if we were to continue on. This means that we will exit the algorithm, calling it

satisfiable and now will use extract assignment algorithm to find a valid assignment.

Since we have the singlton clause x_0 we can set that to 1 (i.e. true). That reduces our other clauses to $(x_1 \vee x_2), (\neg x_1 \vee \neg x_2), (x_1 \vee x_2)$. The C_3 clause is redundant so we can drop it, leaving us with $(x_1 \vee x_2), (\neg x_1 \vee \neg x_2)$. If we arbitrarily set $x_1 = 1$ then $(x_1 \vee x_2)$ goes away and $(\neg x_1 \vee \neg x_2)$ reduces to $(\neg x_2)$. In turn, it is obvious, then to set $x_2 = 0$ so that the clause $(\neg x_2)$ is satisfactory. This leaves us with the satisfying assignment $\varphi(x_0, x_1, x_2) = (1, 1, 0)$. Since x_1 was determined arbitrarily, there are other satisfying assignments, but this is one that works.

3. (Reductions to SAT) In Lecture 13, we saw an efficient algorithm to solve the Maximum Matching problem in Bipartite Graphs. A variant of Maximum Matching is Maximum 3-D Matching, where we are given not a graph $G = (V, E)$ but a "3-uniform hypergraph" $H = (V, E)$, where the *hyperedges* $e \in E$ now are *triples* of vertices. Analogously to restricting to bipartite graphs, we restrict to *tripartite* hypergraphs where V is the union of 3 disjoint sets $V_0 \cup V_1 \cup V_2$ and every hyperedge $e \in E$ has exactly one vertex from each of V_0, V_1, V_2. For example, hyperedges may consist of compatible patient-donor-timeslot triples (if there are only certain timeslots in which a patient and donor are available for a surgery), or compatible surfer-surfboard-fins triples (if each surfer only likes to ride certain surfboards with certain fins installed). Now the goal is to find a maximum-sized set $M \subseteq E$ such that every vertex $v \in V$ is contained in at most one edge of M. Unfortunately, unlike matching in graphs, there is no polynomial-time algorithm known for Maximum 3-D Matching, even if we restrict to tripartite hypergraphs and even if we only want to find *complete matchings* — ones that match every vertex in V_0 (i.e. we are looking for a matching of size $|M| = |V_0|$):

Input	**:** Three disjoint finite sets of vertices V_0, V_1, V_2 and a set E of hyperedges such that each $e \in E$ contains exactly one vertex from each of V_0, V_1, and V_2.
Output	**:** A set M of hyperedges such that every vertex is in at most one hyperedge in M, and every vertex in V_0 is in a hyperedge in M (if one exists).

Computational Problem 3dCompleteMatching

(a) Show that there is a polynomial-time reduction of 3dCompleteMatching to SAT that, on hypergraphs with $n = |V_0| + |V_1| + |V_2|$ vertices and $m = |E|$ hyperedges, makes one oracle query on a formula with m variables and $O(m^2)$ and runs in time $O(m^2)$. Be sure to prove the correctness of your algorithm and analyze its runtime.
Thus, even though the fastest known algorithms for 3dCompleteMatching run in exponential time, we can use SAT Solvers to solve it much more efficiently on many instances that arise in practice.

First lets recognize the high level correctness of this method. If we are able to set up φ correctly then if all its clauses are satisfied with some set of edges m then we are naturally proving a valid matching. This is because if satisfiability is met, and we properly established the clauses of φ to represent the problem, then the matching will work. The critical component, then, is ensuring that φ is properly defined to represent the 3dCompleteMatching problem. Now lets define φ:

We need ensure that every vertex in V_0 is covered exactly once, and ensure that no vertex across all of V_1, V_2 is covered more than once. This is because, as mentioned in the problem, for the output M as a set of hyperedges, every vertex is in at most one hyperedge, and every vertex in V_0 is in a hyperedge. Furthermore we know the size of M to be $|V_0|$ because at most there will be $|V_0|$ hyperedges because every vertex in V_0 must be included, and they cannot be included twice. Now also, lets say that for every hyperedge in E, called x_e where e is from $0 - |E|$, x_e is either true or false depending on whether it is included in M or not. Then if there is a valid matching all the clauses will be met by x_e being true if $x_e \in M$ or false if $x_e \notin M$. For the sake of this problem we will say that for vertex v, $E(v)$ represents the edges that it is a part of.

For V_0 we have to first set it up so that every vertex in V_0 is included at least once. That means that for every v in V_0, $\bigvee_{e \in E(v)} x_e$. This is a series of ors that checks to see whether any of the edges included with v ($E(v)$) is also a hyperedge in M ($x_e = true$). If any are in M, the clause will return true. This ensures every vertex in V_0 is used at least once.
Now we ensure every v in V_0 are not included more than once because since the above clauses guaranteed every vertex be included, we need now to cap them to strictly 1. To do that, we see that every pair of edges included in $E(v)$ do not both occur. So for every $e, e' \in E$, where $e \neq e'$ we check $\neg x_e \vee \neg x_{e'}$. You do it in pairs, because the clause is violated if both are x_e and $x_{e'}$ are present, and you have to check whether that happens for every pair option in $E(v)$. This safeguards against any vertex being present in multiple hyperedges, which is necessary for a matching problem.
On the other two sets of vertices, V_1 and V_2, we only need to ensure that they are not used more than once. Unlike V_0 where they had to be used exactly once V_1 and V_2 vertices maybe used zero or one times. To check for this we utilize a similar set of clauses that capped V_0 to a max of one use. So every vertex $v \in V_0 \cup V_1$ is included in at most 1 hyperedge in M. This is done by setting up a clause $\neg x_e \vee x_{e'}$ for each pairing $e, e' \in E(v)$ where $e \neq e'$.

In summary, there are 4 sets of clauses (2 for V_0 and 1 for each V_1 & V_2):

i. *Every vertex in V_0 is used: $\forall v \in V_0$, $\bigvee_{e \in E(v)} x_e$. Which represents every vertex in V_0 having one of its edges ($E(v)$) be in M.*
ii. *Every vertex in V_0 being used no more than once: $\forall v \in V_0$, $\neg x_e \vee x_{e'}$ for each pairing $e, e' \in E(v)$ where $e \neq e'$.*
iii. *Every vertex in V_1 being used no more than once: $\forall v \in V_1$, $\neg x_e \vee x_{e'}$ for each pairing $e, e' \in E(v)$ where $e \neq e'$*
iv. *Every vertex in V_2 being used no more than once: $\forall v \in V_2$, $\neg x_e \vee x_{e'}$ for each pairing $e, e' \in E(v)$ where $e \neq e'$*

A singular clause is $O(1)$ time. Thus to check the total runtime we check the number of clauses used. We also know that there will be m variables because as defined x_i represents the ith hyperedge in E and is either true or false depending on whether $x_i \in M$. Since i can go from 0 to m, because $m = |E|$ then there will be m variables. Constructing each clause in item (i) above takes $O(m)$ time, with m variables. Each subsequent set of

clauses above is of size $O(m^2)$ clauses. This is because each vertex contributes to at most $O(m)$ pairs of hyperedges, and there are at most m total vertices $m * O(m) = O(m^2)$. Finally, the oracle to the SAT solver is $O(1)$ time so we are left with $O(m^2)$ time.
Further proof of correctness:
Let first prove item i above. These sets of clauses are only satisfied $\iff$ every vertex in V_0 is in M. Lets say it is the clause is satisfied, but lets also say for sake of contradiction that there is not a vertex v_j in M, and $v_j \in V_0$. Then $E(v_j)$ is set of edges in which are related to v_j. In this case all variables in CNF $(x_{E(v_j)_1}, \vee ... \vee x_{E(v_j)_k})$ are 0. They are all 0, because none of $E(v_j) = (e_1, ...e_k)$ are included in M, $\Rightarrow$ none of the indicator variables $x_{E(v_j)} = 1$. If they are all 0, then it is a contradiction that the clause is satisfied. In the other direction, if the clause is satisfied, and all the vertices are included then for every vertex, at least one $E(v) = (e_1, ...e_k)$ will be in M, making at least one x_e in $\bigvee_{e \in E(v)} x_e$ true.
For items ii-iv the proof is the same: These clauses are only satisfied if at the vertex is used at most once in M, that is it cannot be used more than once. Lets say for sake of contradiction the clause is satisfied but v_j is also included twice. That means there will 2 edges e_i, e_j off of v_j (i.e. in $E(v_j)$) where $e_i \neq e_j$, that are included in M. As a result, x_i, x_j representing indicator for e_i, e_j being in M, respectively, are true. Since the clauses for ii-iv will find all such $(\binom{m}{2})$ edge pairings in M, it will create the clause $(\neg x_i \vee \neg x_j)$. However, both are true so the clause simplifies to $(\neg 1 \vee \neg 1) = (0 \vee 0)$ which is unsatisfactory. This then contradicts that it is possible for a vertex to be included more than once and the clauses to be satisfied. In the other direction, if every vertex is included at most once then every pairing of $e_i, e_j \in E(v)$ translated to x_i, x_j will either be of type $(0,0), (0,1), (1,0)$. Furthermore there will be only one pairing $(0,1)_1$ and one $(1,0)_2$ because in the respective cases $x_{j_1} = x_{i_2}$. As such in the case $(0,0)$ for clause $(\neg x_i \vee \neg x_j)$ you get $(\neg 0 \vee \neg 0) = (1 \vee 1)$ which is satisfactory. Or $(1,0) \rightarrow (\neg 1 \vee \neg 0) = (0 \vee 1)$ which is satisfactory. And finally $(0,1) \rightarrow (\neg 0 \vee \neg 1) = (1 \vee 0)$ which is satisfactory. In this case there will be no two edges in M which share a vertex so there will only ever be at most one of $x_i, x_j \in \binom{m}{2}$ permutations of $e_i, e_j \in E(v)$ that is equal to 1. As shown above when at most one is equal to 1, then the clauses succeed.
As such since both types of clauses succeed in setting up the limitations of the problem, and SAT solvers can correctly find satisfying assignments to the clauses, we have shown proof of correctness.

(b) (optional) Come up with a polynomial-time reduction for the version of 3dMatching where we are also given a number $k \in \mathbb{N}$ as part of the input and want to find a matching of size k (rather than one of size $|V_0|$). (Hint: you will probably want to use more than m boolean variables, at least $k \cdot m$, possibly more, depending on how you approach the problem.)

4. (Reflection): Describe two concrete ways in which you have supported, or will try to support, your classmates' learning in the course since the last time we asked this question (ps2). Be specific, connecting your answer to the structure of cs1200.

   Note: As with the previous psets, you may include your answer in your PDF submission, but the answer should ultimately go into a separate Gradescope submission form.

   Quick note on grading: Good responses are usually about a paragraph, with something like 7

or 8 sentences. Most importantly, please make sure your answer is specific to this class and your experiences in it. If your answer could have been edited lightly to apply to another class at Harvard, points will be taken off.

Some specific ways I have supported my classmates learning in this course have been trying to talk out solutions. I have two teammates on the soccer team in this class, and we often find ourselves talking about problems on the pset on the way to practice. I find it helpful to talk out the problems because I can help clarify their misunderstandings while also see where there are gaps in my knowledge. One specific time that I remember this happening was talking specifically about problem 2b from PS5. He did not understand why it would be true for there to be a maximally independent set. However, after talking it came down to just a misinterpretation of the problem as he had thought that every coloring would be a maximal independent set. Had the conversation not happened, he would have went on thinking about the problem wrong. I know it is a minor correction, and minimal help, but even small conversations like these helped him grasp the material more fully, and allowed me to practice explaining the solution aloud. Going forward, I hope to keep talking out problems with them as we even talked out problem 2 for this problem set as I was worried I was doing it wrong at one point.

5. Once you're done with this problem set, please fill out this survey so that we can gather students' thoughts on the problem set, and the class in general. It's not required, but we really appreciate all responses!