CS 341 - Algorithms, Spring 2021. University of Waterloo

Lecture 20: Hard partitioning problems

We will see that the 3-dimensional matching problem and the subset-sum problem are NP-complete.

We then end with some discussions.

3-Dimensional Matching [KT 8.6]

This is a generalization of the bipartite matching problem.

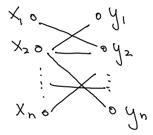
3-dimensional matching (3DM)

Input: Disjoint sets X, Y, Z each of size n, a set $T \subseteq X \times Y \times Z$ of triples.

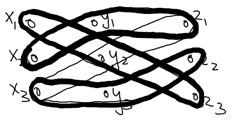
Output: Does there exist a subset of n triples in T so that each element of $X \cup Y \cup Z$ is contained in exactly one of the triples?

The bipartite matching problem is a special case when we only have X, Y and pairs.

bipartite matching



3DM



Input: (x_1, y_2, z_3), (x_2, y_1, z_1), (x_3, y_2, z_1), (x_3, y_3, z_2)

Output: (x_1, y_2, z_3), (x_2, y_1, z_2), (x_3, y_3, z_2) is a perfect matching

A set of n triples is a perfect 3D-matching if every element is contained in exactly one of these triples.

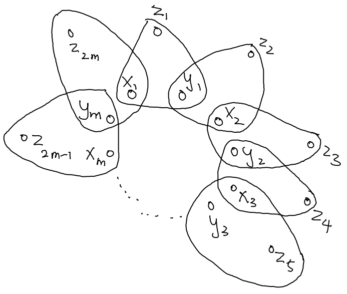
Theorem 3DM is NP-Complete.

Proof Clearly 3DM is in NP. We show that 3DM is NP-Complete by proving 3-SAT $\leq_p$ 3DM.

Given a 3-SAT instances with n variables $x_1, x_2, \ldots, x_n$ and m clauses $C_1, C_2, \ldots, C_m$, we would like to construct an 3DM instance so that the formula is satisfiable iff there is a perfect 3D-matching.

As in the Hamiltonian cycle problem, we would like to create some variable gadgets to capture the binary decisions of the variables.

For each variable v_i, we create the following gadget.

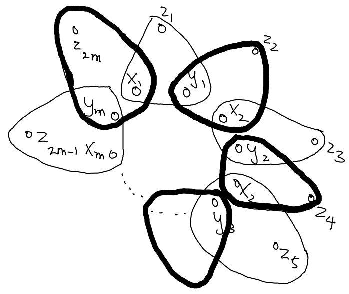


There are m vertices $x_1, \ldots, x_m$, m vertices $y_1, \ldots, y_m$, and $2m$ vertices $z_1, \ldots, z_{2m}$ created for the variable v_i.

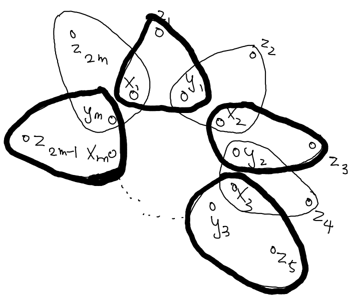
There is a triple x_i, y_i, z_{2i-1} for $1 \leq i \leq m$, and a triple x_{i+1}, y_i, z_{2i} for $1 \leq i \leq m-1$ and a triple x_1, y_m, z_{2m}.

In our construction, we will ensure that the vertices x_i, y_j are not contained in any other triples, besides the two triples in the variable gadget.

This implies that there are only two possibilities to choose the triples in the variable gadget:



Corresponds to setting the variable to be True

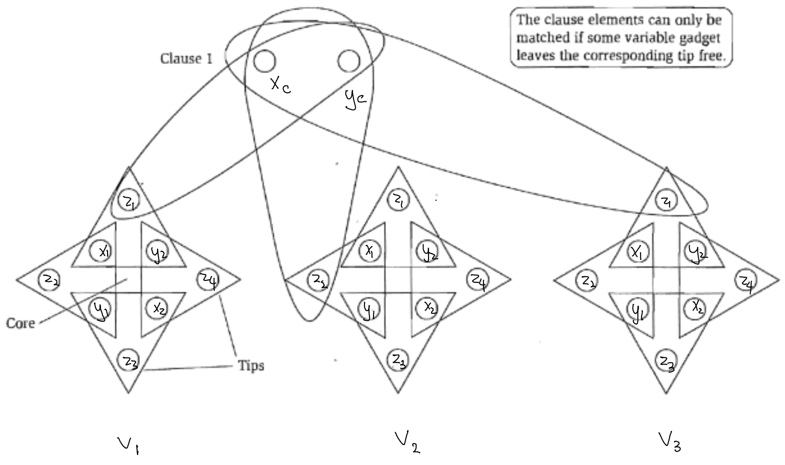


Corresponds to setting the variable to be False

To cover y_1, either we choose x_1, y_1, z_1 or x_2, y_1, z_1. Once this is decided, the remaining decisions are forced.

This captures the binary decision for each variable, as we have one gadget for each variable.

It remains to add some clause structures to the 3DM-instance so that only satisfying assignments "survive".



The clause elements can only be matched if some variable gadget leaves the corresponding tip free.
Clause 1
x_c
y_c
Core
Tips
z_1
x_1
y_2
z_2
z_4
y_1
x_2
z_3
V_1
V_2
V_3

Say we have a clause $C_j = (v_1 \vee \bar{v}_2 \vee v_3)$.

We create two new vertices x_c, y_c for C_j.

If a variable appears as v (but not $\bar{v}$),

we add a triple x_c, y_c, z_{2j-1}

Otherwise, if a variable appears as $\bar{v}$,

we add a triple x_c, y_c, z_{2j}.

We do the same for each clause. Note that the triples for different clauses are disjoint, because they use different "tips" in the variable gadgets.

Each clause can cover one tip. There will be $2nm - m = (2n-1)m$ uncover tips left over.

We will create $(2n-1)m$ pairs of "dummy" vertices x', y', and for each dummy pair, we add a triple (x', y', z) for every tip z in every variable gadget.

Totally, we will add $2(2n-1)m$ new dummy vertices and $(2n-1)m \cdot 2mn < 4m^2n^2$ dummy triples.

This is the construction, which is a little wasteful but certainly can be done in polynomial time.

)

It remains to prove that the formula is satisfiable iff there is a perfect 3D-matching.

$\Rightarrow$) Suppose there is a satisfying assignment.

If $v_i = T$, we cover the variable gadget using the even tips, leaving the odd tips free;

otherwise if $v_i = F$, we cover the variable gadget using the odd tips, leaving the even tips free.

Since this assignment is satisfying all the clauses, for each clause c, there is a literal v_i or $\bar{v}_i$ satisfied.

So, there will be a tip available in the variable gadget for v_i, so that we can choose that triple to cover x_c, y_c.

Finally, for the remaining $(2n-1)m$ uncovered tips, we use the dummy triples to cover them all.

This gives us a perfect 3D-matching.

$\Leftarrow$) Suppose there is a perfect 3D-matching.

For each variable gadget, there are only two ways to cover all the internal vertices x_i, y_j.

If these triples don't cover the odd tips, we set the variable to be True; otherwise False.

Since we can cover the clause vertices x_c, y_c, one of the three tips it connects to is free (i.e. not being used by the triples from the variable gadget).

By our construction, this means that the clause is satisfied by that variable.

Since all clause variables are covered, we have a satisfying assignment. $\square$

<u>**Subset-Sum**</u> [KT 8.8]

Input: n positive integers $a_1, a_2, \ldots, a_n$, and an integer K.

Output: Does there exist a subset $S \subseteq [n]$ with $\sum_{i \in S} a_i = K$?

This is a problem about numbers, so some new ideas are needed to connect to previous combinatorial problems.

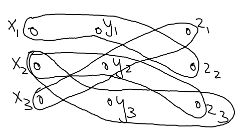
<u>Theorem</u> Subset-sum is NP-complete.

<u>Proof</u> It is clear that Subset-sum is in NP. We prove that it is NP-complete by proving $3DM \leq_p$ Subset-sum.

Given an instance of 3DM, we will construct an instance of subset-sum so that there is a perfect 3D-matching iff there is a subset of certain sum K (whose value is to be determined later).

The idea is quite natural. We will first map a triple to a bit-vector, and then we will map a bit-vector to a number, and so eventually a triple will be mapped to a number.

We first show the reduction using an example and then describe it more formally.



3DM instance with 4 triples (x_1, y_1, z_2), (x_2, y_2, z_3), (x_2, y_3, z_3), (x_3, y_2, z_1) $\longrightarrow$

$$\begin{array}{c|c|c|c|c}
 & v_1 & v_2 & v_3 & v_4 \\
x_1 & 1 & 0 & 0 & 0 \\
x_2 & 0 & 1 & 1 & 0 \\
x_3 & 0 & 0 & 0 & 1 \\
y_1 & 1 & 0 & 0 & 0 \\
y_2 & 0 & 1 & 0 & 1 \\
y_3 & 0 & 0 & 1 & 0 \\
z_1 & 0 & 0 & 0 & 1 \\
z_2 & 1 & 0 & 0 & 0 \\
z_3 & 0 & 1 & 1 & 0
\end{array}$$

Let $|X| = |Y| = |Z| = n$ and let m be the number of triples in the 3DM instance.

We will create a bit-vector for each triple. Each vector has $3n$ coordinates, one for each vertex.

For each triple (x_i, y_j, z_k), we have a bit-vector with 1 in the i-th, $(n+j)$-th, and $(2n+k)$-th coordinates.

From our construction, it is easy to verify that there is a perfect matching in the 3DM instance if and only if there is a subset of vectors that sums to the all-one vector.

In the above example, the vectors v_1, v_3, v_4 sum up to the all-one vector, and the corresponding triples form a 3DM.

We leave the verification of this claim to the reader.

So, now, we have proved that the problem of choosing a subset of 0-1 vectors that sums to $\vec{1}$ is NP-complete.

We would like to reduce this problem to the subset-sum problem to show that it is NP-complete.

A very natural idea is to think of the 0-1 vector as the binary representation of a number.

That is, each triple (x_i, y_j, z_k) is mapped to the number $2^i + 2^{n+j} + 2^{2n+k}$.

With this mapping, if there is a subset of triples that form a perfect 3D-matching, their corresponding numbers would add up to $\sum_{i=1}^{3n} 2^i$, the number that corresponds to the all-one binary string.

However, if there is a subset of numbers that add up to $\sum_{i=1}^{3n} 2^i$, it may not correspond to a perfect 3D-matching.

The problem is that when we add up two numbers both with 1 in the j-th coordinate, the resulting number has a 1 in the $(j+1)$-th coordinate because of "carrying".

This is not our intention, which is to choose a vector with a one in the $(j+1)$-th coordinate.

There is a simple trick to get around this "carrying" problem, so that the above plan would work.

There are at most m numbers.

So, if we choose a large enough base $b = m+1$, there could be no carrying to the next "digit / position".

Final construction: Each triple (x_i, y_j, z_k) is mapped to the number $b^i + b^{n+j} + b^{2n+k}$.

Define $K = \sum_{i=1}^{3n} b^i$, the all-one number in base b (except for the lowest position)

It is clear that this can be done in polynomial time.

Claim There is a perfect 3D-matching iff there is a subset with sum $K = \sum_{i=1}^{3n} b^i$.

$\Rightarrow$) This direction is easy and has been explained in the bit-string setting.

$\Leftarrow$) Since there is no carrying, for each position l in the base-b representation, the subset sum records how many times we have covered the l-th vertex in the 3DM instance.

As the target number K has one in each position, the subset must correspond to a perfect matching.

This claim finishes the proof of the theorem. □

Corollary Knapsack is NP-complete.

Concluding Remarks

Concluding Remarks

First, we summarize what we have done so far.

$$\begin{tikzcd}
 & & \text{all problems in NP} & & & \\
 & & \text{circuit SAT} \arrow[u, "\text{(sketch)}"'] & & & \\
 & & \text{3-SAT} \arrow[u] & & & \\
 & \text{VC} \arrow[ur] \arrow[d, shift left] & \text{HC} \arrow[u] \arrow[dr, shift left] & & \text{3DM} \arrow[ull] & \text{3-coloring} \arrow[ulll, bend right] \\
\text{Clique} \arrow[ur] \arrow[r, shift left] & \text{IS} \arrow[u, shift left] \arrow[l, shift left] & \text{TSP} \arrow[u] & \text{HP} \arrow[ul, shift left] & \text{subset-sum} \arrow[u] & \\
 & & & & \text{knapsack} \arrow[u] &
\end{tikzcd}$$

$\leftarrow$ means $\le_p$

Unlike what we have done in earlier topics, where we showed you some basic examples and asked you to similar or more advanced examples in homework, in NP-completeness, we have showed you the difficult reductions and will ask you to do simple reductions in homework.

Techniques of doing reductions

As we mentioned before, doing reduction requires a different way of thinking.

We need to find a hard problem Y and show that $Y \le_p X$ for our problem X.

It requires practices to search for the right problem Y.

Once you know more NP-complete problems, it will be easier to find a similar problem to do the reduction, e.g. covering type uses VC, partitioning type uses 3DM, etc.

There are three common techniques in proving NP-completeness.

Specialization

Observe that a hard problem is a special case of your problem.

This is the easiest but also most useful technique, as most practical problems are often more complicated with different parameters, and often you may realize that restricting to a simple setting already captures an NP-complete problems.

Some examples that we have seen include TSP and knapsack.

You will see more in Hw5 and supplementary exercises.

Local replacement

This is not as simple as specialization, but not as difficult as gadget design.

We replace each simple structure in problem Y by a simple structure in problem X.

Some examples that we have seen include Circuit-SAT $\leq_p$ 3-SAT where we replaced each logic gate by some simple clauses with the same functionality, DHC $\leq_p$ HC where we replaced each directed edge by an undirected path of length three and connect the paths accordingly, 3DM $\leq_p$ subset-sum which is a more non-trivial example of this kind where we replaced each triple by a number.

You will see more in HW5 and supplementary exercises.

Gadget design

This requires creativity and good understanding to design gadgets and also the plan for the reduction.

We have seen 3SAT $\leq_p$ VC, 3SAT $\leq_p$ DHC, 3SAT $\leq_p$ 3DM, 3SAT $\leq_p$ 3-coloring are all of this kind.

We won't ask this type of questions in HW and in exam.

2 vs 3

It is an interesting phenomenon to observe that 2 is usually easy but 3 is usually hard.

For example, 2SAT is easy (see supplementary exercise list 2) but 3SAT is hard,
2-coloring is easy but 3-coloring is hard, 2D-matching is easy but 3D-matching is hard.

Usually, 2 is easy because once we make a decision then everything else is forced,
while 3 is difficult because after we made a decision we are still left with binary decisions.

Decision problems vs search problems

You may wonder whether restricting to decision problems will limit the scope of our problems, because we are usually interested in finding an optimal solution.

Many search problems can be easily reduced to decision problems.

For example, to find a Hamiltonian cycle, we can delete an edge and ask again whether the graph still has a Hamiltonian cycle, to remove all the edges until we are left with a Hamiltonian cycle.

You are encouraged to try this for other problems, e.g. VC, subset-sum, 3SAT, etc.

Actually, one can prove formally that any NP-complete search problem can be reduced to polynomially many NP-complete decision problems, by reconstructing a solution bit by bit.

So, in terms of polynomial time solvability, this is without loss of generality.

Finally, to determine the optimal value (say for min-VC, max-IS), we can simply do binary search to figure out the optimal value by solving a logarithmic number of decision problems.
