Olympiads in Informatics, 2016, Vol. 10, 87–98

DOI: 10.15388/ioi.2016.06

Understanding Unsolvable Problem

Jonathan Irvin GUNAWAN
Undergraduate Student
School of Computing, National University of Singapore
Computing 1, 13 Computing Drive, Singapore 117417
e-mail: jonathan.irvin@yahoo.com

Abstract. In recent IOIs, there are several problems that seem unsolvable, until we realise that there is a special case to the problem that makes it tractable. In IOI 2014, the problem 'Friend' appears to be a standard NP-hard Maximum Independent Set problem. However, the graph is generated in a very special way, hence there is a way to solve the problem in polynomial time. There were several contestants who didn't identify the special case in this problem, and hence were stuck at the problem. In this paper, we will study a well-known technique called reduction to show that a problem we are currently tackling is intractable. In addition, we introduce techniques to identify special cases such that contestants will be prepared to tackle these problems.

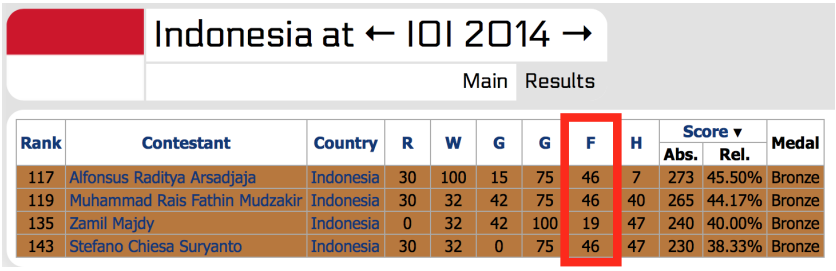
Keywords: special case, unsolvable, NP-hard.

1. Introduction

The problem 'Friend' in IOI 2014 required contestants to find a set of vertices with maximum total weight, such that no two vertices in the set are sharing a common edge. This is a classical Weighted Maximum Independent Set problem. We can show that Weighted Maximum Independent Set problem is NP-hard by reduction from 3-SAT (Cormen *et al.*, 2009). Since the formulation of NP-completeness 4 decades ago, no one has been able to propose a solution to any NP-hard problem in polynomial time. Clearly, it is not expected that a high school student can solve the problem in 5 hours. None of the Indonesian IOI 2014 team solved this problem during the contest. After returning from the competition, I asked the Indonesian team about this problem. None of the team members were aware of the fact that Maximum Independent Set is an NP-hard problem, and thus were stuck trying to solve a general Maximum Independent Set problem.

A similar problem also occurred in IOI 2008. The problem 'Island' required contestants to find a longest path in a graph with 1,000,000 vertices. The longest path problem is a classic NP-hard problem which can be reduced from the Hamiltonian path problem. If a contestant is not aware that the longest path problem is difficult to solve, the contestant may spend a lot of his/her time just to tackle the general longest path problem, without realising that there is a special case to the given graph.

Generally, some contestants spend too much thinking time trying to solve something that is believed to be unsolvable. If only they realise that their attempt is intractable, they may try a different approach and find a special case of this problem. In section 2 of this paper, we will introduce a classic reduction technique often used in theoretical computer science research. In the context of competitive programming, we may find out that a problem which we are attempting is unlikely to be solvable. After realizing that a problem is intractable, we are going to discuss how to proceed to solve the problem in section 3. Finally, in section 4 we will take a look at some common special cases in competitive programming that can be used to solve these kind of problems.

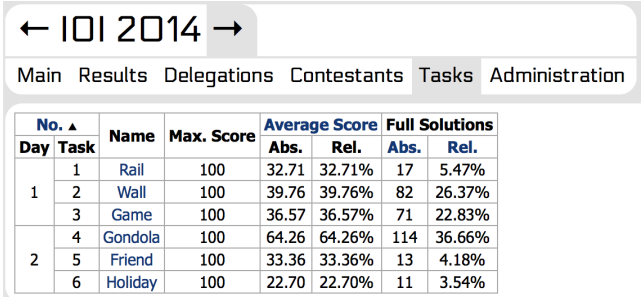


Indonesia at ← IOI 2014 →

Main Results

Rank	Contestant	Country	R	W	G	G	F	H	Score ▾		Medal
									Abs.	Rel.	
117	Alfonsus Raditya Arsadjaja	Indonesia	30	100	15	75	46	7	273	45.50%	Bronze
119	Muhammad Rais Fathin Mudzakir	Indonesia	30	32	42	75	46	40	265	44.17%	Bronze
135	Zamil Majdy	Indonesia	0	32	42	100	19	47	240	40.00%	Bronze
143	Stefano Chiesa Suryanto	Indonesia	30	32	0	75	46	47	230	38.33%	Bronze

Fig. 1. The result of Indonesian team in IOI 2014, taken from `http://stats.ioinformatics.org`. The red squared column highlights the 'Friend' problem.

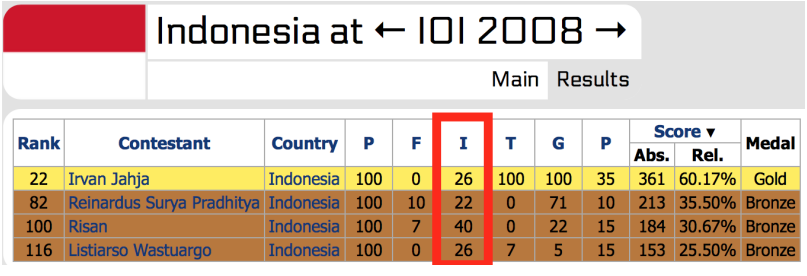


← IOI 2014 →

Main Results Delegations Contestants Tasks Administration

No. ▲		Name	Max. Score	Average Score		Full Solutions	
Day	Task			Abs.	Rel.	Abs.	Rel.
1	1	Rail	100	32.71	32.71%	17	5.47%
	2	Wall	100	39.76	39.76%	82	26.37%
	3	Game	100	36.57	36.57%	71	22.83%
2	4	Gondola	100	64.26	64.26%	114	36.66%
	5	Friend	100	33.36	33.36%	13	4.18%
	6	Holiday	100	22.70	22.70%	11	3.54%

Fig. 2. IOI 2014 tasks statistics, taken from `http://stats.ioinformatics.org`. 'Friend' problem is the second least accepted problem in IOI 2014. It may be because some contestants (at least all the Indonesians) were stuck at trying to solve a general case of Maximum Independent Set.



Indonesia at ← IOI 2008 →

Main Results

Rank	Contestant	Country	P	F	I	T	G	P	Score ▾		Medal
									Abs.	Rel.	
22	Irvan Jahja	Indonesia	100	0	26	100	100	35	361	60.17%	Gold
82	Reinardus Surya Pradhitya	Indonesia	100	10	22	0	71	10	213	35.50%	Bronze
100	Risan	Indonesia	100	7	40	0	22	15	184	30.67%	Bronze
116	Listiarso Wastuargo	Indonesia	100	0	26	7	5	15	153	25.50%	Bronze

Fig. 3. The result of Indonesian team in IOI 2008, taken from `http://stats.ioinformatics.org`. The red squared column highlights the 'Island' problem.

2. Identifying Intractability of a Problem through Reduction

We would like to know that the problem that we are attempting is unlikely to have an immediate solution. The most common way is to apply a well-known technique called reduction. Suppose we know that problem X is impossible to solve, and we also know that we can solve problem X by using problem Y as a black-box[1]. If we can solve problem Y, then we can solve problem X as well. Therefore, problem Y is also impossible to solve.

We are using NP-hard problems for illustration. Recall that NP-hard problems have yet to be solved in polynomial time for more than 4 decades. MIN-VERTEX-COVER is a graph problem that involves finding a minimum subset of nodes such that for every edge, at least one of its endpoint is in the subset. MAX-INDEPENDENT-SET is a graph problem of finding a maximum subset of nodes such that for every edge, at most one of its endpoint is in the subset. Suppose we already know that MIN-VERTEXCOVER is a NP-hard problem. Therefore, we can show that MAX-INDEPENDENT-SET is also a NP-hard problem by reducing a MIN-VERTEX-COVER problem into a MAX-INDEPENDENT-SET problem.

We will first prove the following two lemmas.

Lemma 1. *If $S \subseteq V$ is an INDEPENDENT-SET of graph $G(V, E)$, then $V - S$ is a VERTEX-COVER of the graph $G(V, E)$.*

Proof. Let us assume that $V - S$ is not a VERTEX-COVER. Therefore, there are two vertices $A, B \notin V - S$ and there is an edge connecting A and B. Since $A, B \notin V - S$, we have $A, B \in S$. As A and B are connected by an edge, we note that S is not an INDEPENDENT-SET. This is a contradiction. Therefore $V - S$ is a VERTEX-COVER.

Lemma 2. *If $S \subseteq V$ is a VERTEX-COVER of graph $G(V, E)$, then $V - S$ is an INDEPENDENT-SET of the graph $G(V, E)$.*

Proof. The proof is actually similar to the previous lemma. Let us assume that $V - S$ is not a INDEPENDENT-SET. Therefore, there are two vertices $A, B \in V - S$ and there is an edge connecting A and B. Since $A, B \in V - S$, we have $A, B \in S$. As A and B are connected by an edge, we note that S is not a VERTEX-COVER. This is a contradiction. Therefore $V - S$ is an INDEPENDENT-SET.

Theorem 1. *If $S \subseteq V$ is a MAX-INDEPENDENT-SET of graph $G(V, E)$, then $V - S$ is a MIN-VERTEXCOVER of the graph $G(V, E)$.*

Proof. We know that $V - S$ is a vertex cover by lemma 1. The only thing that remains for us to prove is its minimality. Suppose $V - S$ is not minimum vertex cover. Then, there is another vertex cover $V - S'$ where $|V - S'| < |V - S|$, which implies that $|S'| > |S|$. By lemma 2, S' is an independent set. Therefore, S is not a maximum independent set. This is a contradiction. Therefore $V - S$ is the minimum vertex cover.

[1] We say that problem X is reducible to problem Y

From the above theorem, we conclude that a MIN-VERTEX-COVER can be easily constructed if we have a MAX-INDEPENDENT-SET.

In the beginning of this section, we assume we know that MIN-VERTEX-COVER is a NP-hard problem. It is good to know as many NP-hard problem as possible. This is necessary so that if we encounter a new problem X, we can use any of the NP-hard problems that we know, reduce it to problem X, and thus prove that X is also NP-hard.

3. How to Proceed

Suppose we already know that a problem is unsolvable (i.e. any known algorithm will not solve this problem in time). In competition, it is impossible to complain that "This is unsolvable, can you eliminate this problem?" to the judges, since the judges believe they have a solution. Such a request is absurd when there are already several contestants who have solved that problem. Also, in a major competition (e.g. ACM International Collegiate Programming Contest World Finals, IOI), it is unlikely that the judges have incorrect solution.

3.1. *Approximation*

In real life, when we cannot find the optimal solution, we can try to find the solution that is close to the optimal solution. More specifically, we try to find a solution that is not larger than α (where $\alpha > 1$) times the optimal solution for a minimisation problem. The most common approximation algorithm I found in textbooks is the 2-approximation MIN-VERTEX-COVER problem, which means that the algorithm will not choose more than twice the number of vertices than the optimal solution. However, approximation problems rarely occur in competitive programming (especially IOI). One of the reason is because to create this kind of problem, the judges have to know the optimal solution in order to verify that the contestant's solution is indeed α-approximation. However, generating the optimal solution is impossible (or takes a long time). Since this approach is not really suitable for competitive programming, I will not discuss this approach in detail.

3.2. *Pruning*

This approach is useful in some competitive programming problems. In ACM International Collegiate Programming Contest (ICPC) World Finals 2010, problem I (Robots on Ice) required the contestant to count the number of Hamiltonian Paths with constraints (ACM ICPC World Finals 2010 problem statement, n.d.), which is known to be a NP-hard problem. While finding all possible paths is impossible, the solution

for this problem is to prune the exponential algorithm we use to find all possible paths (ACM ICPC World Finals 2010 Solutions. n.d.). If at some point we know that it is impossible to visit the rest of the unvisited points, then we can prune the path and backtrack immediately. However, it is not very suitable for IOI. Usually, IOI problems require deep analysis from the contestant. It is rare that we can get Accepted by only "hacking" a complete search algorithm. Therefore, we will not discuss this technique in detail.

3.3. *Finding Small Constraints*

Suppose there is an NP-hard problem with large N that is impossible to solve exponentially (e.g. $N > 50$). Sometimes we should also look for other small constraints that may help. For example, a SUBSET-SUM problem is considered an NP-hard problem, and will not be solvable with $N = 100$. When we are given the constraint that all the elements inside the array are small (e.g. $[0, 100]$), this problem can be solved using $O(NX^2)$ Dynamic Programming, where X is the upper bound of the elements inside the array. Even though the running time of the algorithm is exponential to the size of the input, the algorithm is still fast enough for the given constraint. Another way to apply this technique is when the value of N is not too large (e.g. $N < 40$). For example, while $O(2^N)$ algorithm for $N = 36$ is unlikely to run in one second, we can, for instance, use a $O(2^{N/2})$ Meet In The Middle algorithm for solving a problem like SUBSET-SUM. While $O(2^{N/2})$ is still exponential to N, it is much faster than $O(2^N)$, and the range of N that it can solve is twice the range of N using a $O(2^N)$ solution.

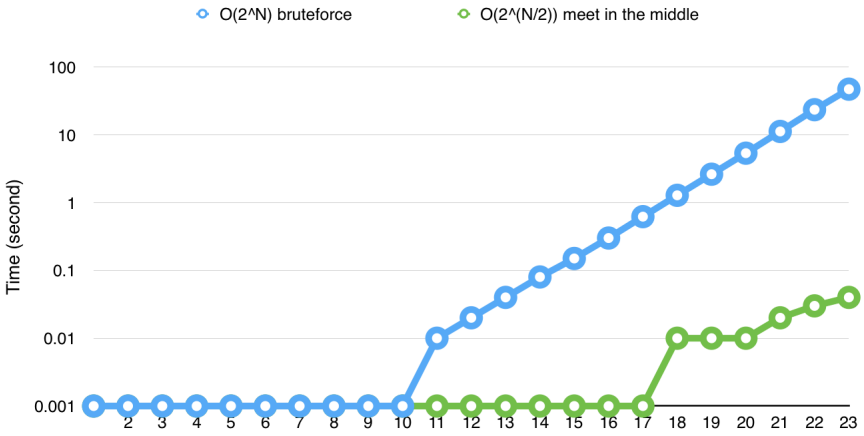


Fig. 4. Comparison of $O(2^N)$ and $O(2^{N/2})$ SUBSET-SUM algorithm running time for various input sizes. This experiment is run 100 times for each value of N on a MacBook Pro (Retina, 13-inch, Early 2015).

3.4. *Finding Special Cases*

This is the most suitable approach in IOI, and thus is the main focus of this paper. To solve IOI 2008 Island and IOI 2014 Friend, we need to use this approach. We must find a special constraint in the problem such that this constraint allows the problem to be solvable in polynomial time. We can check whether an additional constraint causes a problem to be solvable in polynomial time using the reduction proof of the original problem (without the additional constraint), and check whether the proof still holds given the additional constraint to the problem.

We will give we one example of a special case in a NP-hard problem. Suppose we have a function with the following formula

$$f(n) = \begin{cases} 1, & n = 1 \vee n = 2 \\ f(n-1) + f(n-2), & n > 2 \end{cases}$$

We consider the sequence $F = \{f(n)\}_{n=1}^{\infty}$, and we define $F(N)$ to be the first N terms of F. We want to know whether we can create a partition of $F(N)$ into two disjoint multisets A and B such that the sum of all elements in A is equal to the sum of all elements in B.

This looks like a classic PARTITION problem. PARTITION problem is NP-hard by reduction from SUBSET-SUM. Therefore, for a large value of N, it is unlikely to be able to find an algorithm that finds A and B in an efficient way. However, this sequence is defined in a very special way, in the sense that F is defined using the aforementioned recurrence. Therefore, we should inspect the recurrence formula more closely.

Lemma 3. *Any consecutive subsequence of F with length multiples of three can be partitioned into two multisets of equal sum.*

Proof. Pick any consecutive subsequence of F with length multiples of three, which we will denote by $F' = \{f(a), f(a+1), f(a+2), ...f(b)\}$ for some $a < b$. We can partition F' into

$$A = \{f(a+3k), 0 \leq k \leq \frac{b-a-2}{3}\} \cup \{f(a+1+3k), 0 \leq k \leq \frac{b-a-2}{3}\}$$

$$B = \{f(a+2+3k), 0 \leq k \leq \frac{b-a-2}{3}\}$$

A and B will have the same sum, as for every $0 \leq k \leq \frac{b-a-2}{3}$

$$f(a+3k) + f(a+1+3k) = f(a+2+3k)$$

by the construction of the function.

Theorem 2. *If* N *is divisible by three, then* $F(N)$ *can be partitioned into two multisets of equal sum.*

Proof. If N is divisible by three, then $F(N)$ is a prefix of F with length multiples of three. By lemma 3, $F(N)$ can be partitioned into two multisets of equal sum.

Theorem 3. *If* $N \equiv 1 \pmod 3$, $F(N)$ *cannot be partitioned into two multisets of equal sum.*

Proof. Suppose $F'(N) = F(N) - f(1)$. Note that $F'(N)$ contains $N - 1$ elements. Since $N \equiv 1 \pmod 3$, we have $N - 1 \equiv 0 \pmod 3$. By lemma 3, $F'(N)$ can be partitioned into two multisets of equal sum. Therefore, the sum of all elements in $F'(N)$ is even. However, the sum of all elements in $F(N) = F'(N) + f(1)$, which is odd because $F'(N)$ is even while $f(1)$ is odd. Therefore, there is no way to partition $F(N)$.

Theorem 4. *If* $N \equiv 2 \pmod 3$. $F(N)$ *can be partitioned into two multisets of equal sum.*

Proof. Assign $f(1)$ to A and $f(2)$ to B. Since $f(1) = f(2)$, we are now trying to partition $F'(N) = F(N) - f(1) - f(2)$. $F'(N)$ will have $N - 2$ elements. Since $N \equiv 2 \pmod 3$, we obtain $N - 2 \equiv 0 \pmod 3$. By lemma 3, $F'(N)$ can be partitioned into two multisets of equal sum, which implies our theorem.

Therefore, solving this problem is reduced to checking whether $N \equiv 1 \pmod 3$. We can solve this in $O(1)$.

We will provide more examples of special cases in the following section.

4. Some Example of Special Cases

We will look at some common examples of special cases that may occur in competitive programming problems.

4.1. *Planar Graphs*

Planar graph is a graph that can be drawn on a flat surface without having two edges crossing each other (West *et al.*, 2001). There are many graph problems which are easy to solve if the graph is planar. We will provide several examples.

4.1.1. *Number of Edges*

In simple general graph, the number of edges can be up to a quadratic order with respect to the number of vertices (i.e. $O(V^2)$). This is not the case for planar graph. In a planar graph, we may show that $E \leq 3V - 6$ holds for $V \geq 3$ by using the Euler's formula. Therefore, the number of edges is $O(V)$. Naturally, any algorithm that has $O(E)$ in its running time can be changed into $O(V)$. Computing shortest path in a general graph us-

ing Bellman-Ford algorithm takes $O(VE)$ (Halim and Halim, 2013), but it only takes $O(V^2)$ in a planar graph. Counting the number of connected components using DFS in a general graph takes $O(V + E)$, but it only takes $O(V)$ in a planar graph. Therefore, if the problem requires us to compute the number of connected components in a planar graph, even though the constraint states that $V \leq 100,000, E \leq 100,000^2$, the standard DFS solution still runs under one second (in competitive programming, we assume that 1 million operations can be done in 1 second (Halim and Halim 2013)).

4.1.2. *Maximum Clique Problem*

The Maximum Clique problem requires us to find the maximum set of vertices in a graph, such that every pair of vertices in the set is directly connected by an edge. By reduction from vertex cover, the maximum clique problem on general graph is NP-hard. However, it is easy to solve this problem in planar graph. Consider the problem of colouring a graph such that no two adjacent vertices have the same colour. Note that if there is a clique of size k in a graph, the set of the vertices inside the clique must be coloured with k colours. Therefore, the entire graph requires at least k colours. By the four colour theorem, any planar graph can be coloured with at most four colours. (Gonthier, 2005). Since at most four colours are required to colour a planar graph, there does not exist a clique with more than four vertices. Hence, we can solve the problem by checking whether there is a clique with four vertices. If there is no clique with four vertices, we check whether there is a clique with three vertices. Finding whether there is a clique with k vertices can be solved naively in $O(V^k)$. Therefore, the whole solution takes $O(V^4)$ time, which is polynomial. The solution can be improved to $O(V^2)$. Instead of having four nested loops to find four vertices independently, we can have two nested loops over the edges instead and test whether the four vertices (which are the endpoint of the two edges) form a clique. This solution takes $O(E^2)$, which is the same as $O(V^2)$ in planar graph.

4.1.3. *Problem Example*

We will use a past competitive programming problem to illustrate the importance of the properties in a planar graph. The 'Traffic' problem from Central European Olympiad in Informatics (CEOI) 2011 illustrates this concept well. In this problem, there is a directed graph with up to $V = 300,000$ vertices given in a 2D plane. There are two vertical lines denoted as 'left' and 'right'. All vertices are contained within the 'left' and 'right' lines, with some vertices possibly lying on these two lines. The problem requires the contestant to print the number of visitable[2] vertices lying on the 'right' line from every vertex lying on the 'left' line (which we shall call 'left' vertices and 'right' vertices for simplicity). An instance for this problem can be seen on Fig. 5.

The brute force solution for this problem is to run DFS from every 'left' vertex which gives us a running time of $O(V^2)$. It is very difficult to find a solution (if any) faster than $O(V^2)$ for this problem. However, there is an important constraint on the problem. The graph given in this problem is always a planar graph. We first relabel the 'right' vertices in ascending order according to the *y*-coordinate. The planar properties ensures that for

[2] vertex *v* is visitable from vertex *u* if there is a path from *u* to *v*

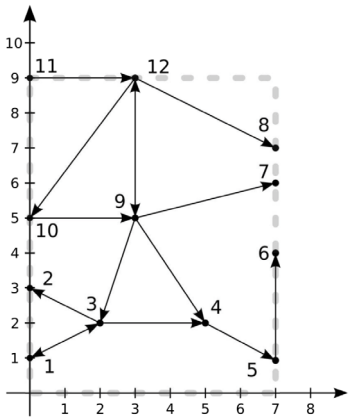


Fig. 5. An instance of problem 'Traffic'. In this example, the expected output is {4, 4, 0, 2}, since the top 'left' vertex can visit 4 'right' vertices, the second top 'left' vertex can visit 4 'right' vertices, the third top 'left' vertex cannot visit any 'right' vertex, and the bottom 'left' vertex can visit 2 'right' vertices.

every 'left' vertex, the sequence of 'right' visitable vertices from that 'left' vertex is a contiguous sequence, assuming that we have removed all 'right' vertices which are not visitable from any 'left' vertex. With this property, there is a $O(V \log V)$ solution (CEOI 2011 Solutions, n.d.).

4.2. *Bipartite Graph*

Bipartite graph is a graph in which the vertices can be partitioned into two disjoint sets U and V such that all edges connect a vertex from U and a vertex from V. Some problems have a bipartite graph as an input although the problem statement does not explicitly state that the given input graph must be bipartite. The problem that we used as an introduction for this paper, IOI 2014 Friend is a very good example. The construction of the graph in subtask 5 of this problem implicitly ensures that the final graph will always be bipartite. There are several graph problems that are NP-hard for general graph but solvable in polynomial time if the graph is bipartite. Since bipartite graph and bipartite matching was recently included in IOI 2015 syllabus (Forišek, 2015), we can expect that this type of problem may be conceived in the near future of IOI. We will take a look at several examples.

4.2.1. *Vertex Cover and Independent Set (and Maximum Matching)*

As we discussed in an earlier section, both of the MIN-VERTEX-COVER and MAX-INDEPENDENTSET problems are NP-hard. However, both of these problems are solvable in polynomial time on bipartite graph. By Konig's theorem, the size of the minimum vertex cover in bipartite graph is equal to the size of the maximum bipartite matching (Bondy and Murty, 1976), and the size of the maximum independent set in

bipartite graph is equal to the number of the vertices minus the size of the maximum bipartite matching. Therefore, both problems are equivalent to finding the size of the maximum bipartite matching, which can be solved in $O(V^3)$ time. Finding the size of the maximum matching in bipartite graph using Hopcroft-Karp algorithm or Maximum-Flow algorithm is much simpler to solve, as compared to using Edmonds Blossom algorithm on general graph. This is actually the solution of the 5th subtask of IOI 2014 Friend.

4.3. *Directed Acyclic Graph*

A directed acyclic graph is a directed graph that does not contain any cycle. Similar to planar and bipartite graphs, there are several graph problems that are much easier to solve if the graph is a directed acyclic graph.

4.3.1. *Minimum Path Cover*

MIN-PATH-COVER is a problem that requires us to find the minimum number of vertex-disjoint paths needed to cover all of the vertices in a graph. By a simple reduction from Hamiltonian Path, this problem is NP-hard. A graph has a Hamiltonian Path if and only if we only need one path to cover all of the vertices. However, this problem can be solved in polynomial time for a directed acyclic graph. For a graph $G = (V, E)$, we create a new graph $G' = (V_{out} \cup V_{in}, E')$, where $V_{out} = V_{in} = V$ and $E' = \{(u, v) \in V_{out} \times V_{in} : (u, v) \in E\}$. Then it can be shown by Konig's Theorem that G' has a matching of size m if and only if there exist $|V| - m$ vertex-disjoint paths that cover all of the vertices in G.

4.4. *Miscellaneous*

4.4.1. *Special Case of CNF-SAT Problem*

We will use Google Code Jam 2008 Round 1A 'Milkshake' problem for this example. This problem requires the contestant to find a solution with the minimum number of true variables that satisfy a CNF-SAT problem with up to 2,000 variables (Google Code Jam 2008 Round 1A, 'Milkshake' problem, n.d.). The CNF-SAT is a satisfiability problem given in a conjunctive normal form (i.e. conjunction of disjunction of literals) which was proven to be NP-hard (Cook, 1971). Therefore, it is unlikely that there is an algorithm to solve a CNF-SAT problem with 2,000 variables in less than 8 minutes[3]. However, there is a special property in this problem, in which at most one unnegated literal exists in each clause. Therefore, all clauses can be converted into Horn clauses. With this property, a linear time algorithm exists. (Google Code Jam 2008 Round 1A, 'Milkshake' solution, n.d.).

[3] In Google Code Jam, contestants are given 8 minutes to produce the output upon downloading the input.

5. Conclusion

In conclusion, we can use a well-known reduction technique to prove that a problem that we are currently attempting to solve is impossible (or at least it is very hard such that no people has been able to solve it for more than 40 years). In competitive programming (including IOI), understanding this technique is essential so that we will not be stuck at trying to solve an impossible problem, thus prompting us to find another way to solve the problem. To prove that a problem is NP-hard, it is good to know as many NP-hard problems as possible, so that we can reduce from any one of the problems that we know to the new problem. Some of the classic NP-hard problems include 3-SAT, Vertex Cover, Independent Set, and Subset Sum. After realizing that the problem is NP-hard, we must be able to find the special case that makes the problem solvable. We must be able to find a special property that breaks the reduction proof. Having a lot of practice on these kind of problems will help us to familiarize with the possibilities of a special case. Some of the common special cases include planar, bipartite, and directed acyclic graph.

References

ACM ICPC World Finals 2010 Problem Statement (n.d.). Available via internet:
`http://icpc.baylor.edu/download/worldfinals/problems/2010WorldFinalProblemSet.pdf`
ACM ICPC World Finals 2010 Solutions (n.d.). Available via internet:
`http://www.csc.kth.se/~austrin/icpc/finals2010solutions.pdf`
Bondy, J.A., Murty, U. S. R. (1976). *Graph Theory with Applications*, Vol. 290, London: Macmillan.
CEOI 2011 Solutions (n.d.). Available via internet:
`http://ceoi.inf.elte.hu/probarch/11/ceoi2011booklet.pdf`
CEOI 2011. 'Traffic' Problem (n.d.), available via internet:
`http://ceoi.inf.elte.hu/probarch/11/trazad.pdf`
Cook, S.A. (1971). The complexity of theorem-proving procedures. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. ACM, 151–158.
Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C. (2009). *Introduction to Algorithms (3rd ed.)*. MIT Press.
Forišek, M. (2015). International olympiad in informatics 2015 syllabus. Available via internet:
`https://people.ksp.sk/~misof/ioi-syllabus/ioi-syllabus.pdf`
Gonthier, G. (2005). A computer-checked proof of the four colour theorem. Available via internet:
`http://research.microsoft.com/en-us/people/gonthier/4colproof.pdf`
Google Code Jam 2008 Round 1A, 'Milkshake' Problem (n.d.). Available via internet:
`https://code.google.com/codejam/contest/32016/dashboard#s=p1`
Google Code Jam 2008 Round 1A, 'Milkshake' solution (n.d.). Available via internet:
`https://code.google.com/codejam/contest/32016/dashboard#s=a&a=1`
Halim, S., Halim, F. (2013). *Competitve Programming 3*. `lulu.com`
IOI 2008, 'Island' Problem (n.d.). Available via internet:
`http://www.ioinformatics.org/locations/ioi08/contest/day1/islands.pdf`
IOI 2014, 'Friend' Problem (n.d.). Available via internet:
`http://www.ioinformatics.org/locations/ioi14/contest/day2/friend/friend.pdf`
West, D. B. *et al.* (2001). *Introduction to Graph Theory*, Vol. 2. Prentice hall Upper Saddle River.

J.I. Gunawan is an undergraduate student studying Computer Science in National University of Singapore. He participated a lot of programming contests, including IOI 2012 and 2013 and ACM ICPC World Finals 2014 and 2015. He is also the lead of the Scientific Committee of Indonesian Computing Olympiad team (TOKI) training Indonesian teams for IOI in 2015–2016.