NP-COMPLETE PROBLEMS

Polynomial

"Computers and Intractability: A Guide to the Theory of NP-Completeness," Michael R. Garey and David S. Jonson, W.H. Freeman and Company, New York, 1979, ISBN: 0-7167-1045-5.

Before we start: be reminded of our model of computation: all basic operations take unit time, we have infinite memory at the register level, and every step is deterministic as described in the algorithm.

A polynomial algorithm is "faster" than an exponential algorithm. As n grows a^n (exponential) always grows faster than n^k (polynomial),

i.e. for any values of a and k, after n> certain integer n_0, it is true that $a^n > n^k$.

Even 2^n grows faster than n^{1000} at some large value of n. The former functions are exponential and the later functions are polynomial.

It seems that for some problems we just may not have any polynomial algorithm at all (as in the *information theoretic* bound)! The theory of NPcompleteness is about this issue, and in general the computational complexity theory addresses it.

Solvable Problems

Some problems are even unsolvable/ undecidable algorithmically, so you cannot write a computer program for them.

Example. *Halting problem*: Given an algorithm as input, determine if it has an infinite loop.

There does not exist any general-purpose algorithm for this problem. Suppose (contradiction) there is an algorithm H that can tell if any algorithm X halts or not, i.e., H(X) scans X and returns True iff X does halt.
Then, write

```
P(X):                    // X is any "input algorithm" to P  while (H(X))
{ };     // condition is true only when X terminates
return;                          // P terminates
End.
```

Then, provide P itself as the input X to the algorithm [i.e., P(P)]: what happens?!
H cannot return T/F for input P, there cannot exist such H.
It is equivalent to the truth value of the sentence “*This sentence is a lie*.”

Note (1), we are considering **Problems** (i.e., for all instances of the problem) as opposed to some **instances of the problem**. For some sub-cases you may be able to solve the *halting problem*, but you cannot have an algorithm, which would solve the halting problem for ALL input.

<u>Different **types of problems**</u>: decision problems, optimization problems, …
Decision problems: Output is *True/False*.

Decision Problem <-> corresponding optimization problem.
Example of 0-1 *Knapsack Decision* (KSD) problem:
Input: KS problem + a profit value *p*
Output: Answer the question "does there exist a knapsack with profit □ *p*?"

Algorithms are also inter-operable between a problem and its corresponding decision problem.

Solve a KSD problem *K*, using an optimizing algorithm:
Algorithm-Optimization-KS (first part of *K* without given profit *p*) return optimal profit *o* if *p*□*o* then return *TRUE* else return *FALSE*.

Solve a KS problem *N*, using a decision algorithm:
For (*o* = Sum-of-all-objects-profit; *o*>0; *o* = *o* - *delta*) do // do a *linear* search, for a small *delta*
  If (Algorithm-Decision-KS (*N*, *o*)) then continue
   Else return (the last value of *o*, before failure, in the above step); Endfor.
// a *binary search* would be faster!!

Note (2), The complexity theory is developed over *Decision problems*, but valid for other problems as well. We will often use other types of problems as examples.

<u>**NP-class of solvable problems**</u>

Deterministic algorithms are where no step is random.

If the program/algorithm chooses a step non-deterministically (by some extraneous influence to the algorithm!) such that it is always the *right choice*, then such an algorithm is called **non-deterministic algorithm**. Example, suppose a 0-1 KS backtrack-algorithm always knows which object <u>to pick up next</u> in order to find an optimal profit-making knapsack!

If one has a *polynomial deterministic algorithm* for a problem (e.g., the sorting problem), then the problem is called a **P-class** problem. And, the set of all such problems constitute the ***P-class.***

If one has a *polynomial non-deterministic algorithm* for a problem, then the problem is called a **NP-class** problem.
And, the set of all such problems constitute the ***NP-class.***

It is <u>impossible</u> to check for a problem's being in NP-class this way, because non-deterministic algorithms are impossible to develop, as defined above. So, how can one check for polynomial complexity of such non-existent algorithms?

However, an equivalent way of developing non-deterministic polynomial algorithm is:
when a solution to the problem is provided (as if someone knows what the solution could be!), then that *proposed solution* is checked by that algorithm in *polynomial* time. Such a *proposed solution* is called a <u>certificate</u> to the input probleminstance.

For example: in a KSD problem, given a certificate *knapsack content* check if the total profit is □ *p* or not.
Complexity: calculation of total profit of the given knapsack content is worst case O(*n*), for *n* objects.

For example: for a Hamiltonian circuit problem instance (*find a cycle in a graph over all nodes but without any node being repeated in the cycle*), a *path* is given as a certificate.

An algorithm would go over the certificate path and check if the first and the last nodes are *same*,
and the rest of the path is *simple* (no node is covered twice), then it will check if all nodes in the graph are *covered* in the path, and then, check if all the *arcs* in the path do actually from the input graph. This takes polynomial time with respect to *N* and *E*. Hence HC is a NP-class problem.

Two important points to note:
(1) NP-class problems are sub-set of the *Solvable* problems,
(2) P-class problems are sub-set of NP-class problems (because if you have a deterministic algorithm to solve any problem instance, then that algorithm can be used to check any certificate in polynomial time also).
[DRAW SETS]

Problems belonging to the NP-class have at least exponential algorithms.

A related question is: does there exist any solvable problem that is not NP?
Answer: yes.
Example: non-HC problem (is there *no HC* in a given input graph) does not have any polynomial non-deterministic algorithm.

If a problem is in NP-class its complement (negative problem as the non-HC problem is) is in *Co-NP*.
All Co-NP problems constitute the Co-NP class of problems. P-class is a subset of the intersection of NP and Co-NP sets.

<u>An IMPORTANT question</u> is: can we write a polynomial algorithm for *every* NP-class problem? The answer is: *we do not know*.

From a reverse point of view, we would like to find an example problem that is in the NP-class and whose information-theoretic lower bound is exponential.
Then, we would at least know that P is a proper subset of NP. We do not yet have any such example either.

All we know now is that **P □ NP**.

Question remains: (1) P □ NP? or, P □ NP? [One should find counterexample(s)]

Or, (2) NP □ P, so that P = NP? [One should prove the theorem]

NP-completeness

Summary: Apparently some NP problems are "harder" in a relative sense than the other problems. If you can write polynomial algorithm for any one problem in this 'hard' group, then it can be shown that every NP-class problem will have a polynomial algorithm. This group of problems is called *NP-complete* problems.

The secret lies in the fact that they are all "related" to *all NP-class problems* (!!!) by directed chains of *polynomial transformations* (explained below).

We will explain polynomial transformations first and then come back to the issue of NP-completeness.

Polynomial Transformation

Problem Transformation: some algorithms which take a decision problem X (or rather ANY instance of the problem of type X), and output a corresponding instance of the decision problem of type Y, in such a way that if the input has answer *True*, then the output (of type *Y*) is also *True* and vice versa. [REMINDER: Problem ≡ Input; & Solution ≡ Algorithm that takes any input and produces correct output.]

For example, you can write a problem transformation algorithm from *3-SAT* problem to *3D-Matching* problem (will see later).

Note that the problem transformations are directed.

When a problem transformation algorithm is polynomial-time we call it a **polynomial transformation.**

Existence of a polynomial transformation algorithm has a great **significance** for the complexity issues.

Suppose you have (1) a poly-transformation A_{xy} exists from a (source) problem X to another (target) problem Y, and (2) Y has a poly algorithm P_y , then you can solve any instance of the source problem X polynomially, by the following method.
Just transform any instance of X into another instance of Y first using A_{xy}, and then use Y's poly-algorithm P_y. Both of these steps are polynomial, and the output (*T/F*) from Y's algorithm is valid for the source instance (of X) as well. Hence, the *True/False* answer for the original instance of P_y (A_{xy} (X)) will be obtained in poly-time. This constitutes an indirect poly-algorithm for X, thus making X also belonging to the P-class.
Note, $|A_{xy}(X)|$ is polynomial with respect to |X|. *Why?*

Once again, **note the direction**.
(*You will be amazed with how many practicing computer scientists get confused with this direction*!)

Cook's theorem.

Cook modeled all NP-problems (an infinite set) to an abstract *Turing machine*. Then he developed a poly-transformation from this machine (i.e., all NP-class problems) to a particular decision problem, namely, the *Boolean Satisfiability* (*SAT*) problem.

Significance of Cook's theorem: if one can find a poly-algorithm for SAT, then by using Cook's poly-transformation one can solve all NP-class problems in poly-time (consequently, P-class = NP-class would be proved).

SAT is the historically first identified *NP-hard problem*!

Further significance of Cook's theorem: if you find a poly-transformation from SAT to another problem Z, then Z becomes another NP-hard problem. That is, if anyone finds a poly algorithm for Z, then by using your polytransformation from SAT-to-Z, anyone will be able to solve any SAT problem-instance in poly-time, and hence would be able to solve all NPclass problems in poly-time (by Cook’s theorem).

These problems, which have a chain of poly-transformation from SAT, are called **NP-hard problems**.

If an NP-hard problem also belongs to the NP-class it is called an NPcomplete problem, and the group of such problems are called **NP-complete problems**.
[DRAW SETS]

Significance of NP-hard problems

As stated before, if one finds any poly-algorithm for any NP-hard problem, then
We would be able to write polynomial algorithm for each of NP-class problems, or
NP-class = P-class will be proved (in other words, poly-algorithms would be found for all NP-class problems).

Unfortunately, **neither** anyone could find any poly algorithm for any NPhard problem (which would signify that P-class = NP-class); **nor** anyone could prove an exponential information-theoretic bound for any NP-complete problem, say problem L (which would signify that L is in NP but not in P, or in other words that would prove that P-class □ NP-class). The NP-hard problems are the best candidate for finding such counterexamples. [*There is a claim in Summer 2010 of a lower bound-proof of some NP-hard problem, from IBM research!*]

As a result, when we say a problem X (say, the KSD problem) is NPcomplete, all we mean is that
IF one finds a poly-alg for X, THEN all the NP-class problems would have poly algorithm.
We also mean, that it is UNLIKELY that there would be any poly-algorithm found for X.

P □ NP is a mathematical **conjecture** currently (YET to be proved as a theorem, *see above though*).
Based on this conjecture many new results about the complexity-structure of computational problems have been obtained.
Note this very carefully: NP (as yet in history) does NOT stand for "nonpolynomial."

[Also, note that NP-complete problems do have solutions, but they are all exponential algorithms, so far! A common student-mistake is to confuse NPcomplete problems with unsolvable problems.]

There exist other hierarchies. For example, all problems, which need polynomial memory-space (rather than time) form PSPACE problems.

Polytime algorithm will not need more than poly-space, but the reverse may not be necessarily true. Answer to this question is not known. There exists a chain of poly-transformations from all PSPACE problems to the elements of a subset of it. That subset is called *PSPACE-complete*. PSPACE-complete problems may lie outside NP-class, and may be harder than the NP-complete problems. Example: *Quantified Boolean Formula* (first-order logic) is PSPACE-complete.

NP-completeness: FAQ

(following FAQ may repeat information from above)

Why prove a problem to be NP-hard?

So that, (a) one does not have to spend resources on finding out a polynomial algorithm for the problem, and (b) other problems could be proved to be NP-hard by using this problem.

When to attempt such a proof?

When polynomial algorithms are not found after putting reasonable amount of efforts, or based on other intuitions it appears that the problem in question may be NP-hard.

What are the steps in proving a problem to be NP-complete?

First, try to prove it to be NP-hard, by (1) finding a related problem which is already found to be NP-hard (choosing such a suitable "source" problem close to your "target" problem, for the purpose of developing poly-trans, is the most difficult step), and then (2) developing a truth-preserving polynomial problem-transformation from that source problem to your target problem (you will have to show the transformation-algorithm’s (1) correctness and (2) poly-time complexity).
Significance: if anyone finds poly algorithm for your “target” problem, then by using your poly-trans algorithm one would be able to solve that “source” NP-hard problem in poly-time, or in other words P would be = NP. [Note, truth-preservation in poly-transformation works only for decision problems, for other type of problems you may have to first develop a corresponding

decision problem, whose algorithm could be used for solving the original non-decision problem (e.g., Knapsack has corresponding Knapsack-decision problem).]

Second, try to prove that the given problem is in NP-class: by developing a polynomial algorithm for checking any "certificate" of any probleminstance.

Does there exist any short-cut for proving NP-hardness?

Yes. Actually, many.
Example: take a simpler (restricted) version of your problem and prove it to be NP-hard, then the harder original problem is automatically proved to be NP-hard (again, note the direction carefully).
Say, prove 3-SAT problem to be NP-hard, then the generalized SAT problem would be automatically NP-hard.

What is the significance of a poly-transformation from a problem *X* (where only exponential algorithm is available currently) to a P-class problem *Y* (polynomial algorithm is available) This introduces a new indirect polynomial algorithm for *X*.

How to react if someone claims to have a polynomial algorithm for an NPhard problem?

You could go through a checklist as follows:
(a) First check the correctness of the algorithm *for any problem instance*, it may be actually making some assumption for the input, thus restricting input types – in other words, it is an approximate algorithm (If the algorithm is correct, then)
(b) check if its asymptotic complexity is really polynomial or not, (if polynomial and not even pseudo-polynomial, then)
(c) check the NP-hardness proof of the problem, by verifying the problem transformation's correctness, that has been used in such a proof, (d) check if the above transformation is really polynomial in complexity or not, (if correct, then)
(e) check if the source problem that was originally chosen for the above proof is really an NP-hard problem or not, <<and its back-chain of polytransformations to the SAT problem>>
(f) *if all the above checks succeed, then accept that P=NP has been proved and recommend the claimant for the Turing award!!!*

How to deal with a problem once it is proved to be NP-hard?

It is *unlikely* that there would be any polynomial algorithm for solving the problem in question. So, you could take one of the following approaches (*TAPE*):
(T) Find if there exists a "tractable" **sub-problem**, which is realistic enough for your application, and for which one can develop a polynomial algorithm. [Example: The minimum-tardy task-scheduling problem with each taskduration assumed to be one unit of time, and the greedy polynomial algorithm for it. Another example: 2-SAT]
(A) See if an **approximate** solution to the problem is good enough for the application, when such an approximate answer could be found out with a polynomial algorithm. [Example: The multi-processor scheduling problem's greedy polynomial algorithm with a bounded sub-optimal solution.] **(P)** See if a **pseudo-polynomial** algorithm would work (e.g., 0-1 Knapsack problem's dynamic programming algorithm), or see if one could improve upon an exponential-algorithm through some pruning or other heuristics such that the algorithm works fast enough for practical input (as in the use of bounding functions in the backtracking algorithms, or the use of branch and bound techniques there).

(E) If you have developed an exponential algorithm, possibly with **pruning**, and you think that it works quite efficiently on "most" of the input instances of your domain, then you may like to justify your claim by doing some experiments with your algorithm. Some randomized algorithms fall into this category of approach.

EXAMPLE OF A POLYNOMIAL TRANSFORMATION (SAT to 3-SAT)

SAT Problem, or Boolean Satisfiability problem

Example,
(1) A set of Boolean variables, U = {a, b, c}. (2) A (conjunctive) set of *clauses* each consisting of a (disjunctive) set of *literals* (a variable or its negation), C = {{a, b}, {~a, ~b}, {a, c}, {~a, ~c}}. (3) Question: does there exist any set of assignments for the variables such that the clause is True. A clause is True if the assignment makes at least one literal true in the clause.
Output: An answer: *yes/no* (decision problem).
Answer to the above problem instance: *Yes* (for a=T, b=F, and c=F).

A backtracking algorithm for solving SAT is easy to devise. [DRAW A BACKTRACK TREE]:
Worst-case complexity?

SAT is in NP-class: given a certificate truth assignment you can trivially check if it leads to the "Yes" answer or not, and you can do that in linear time O(*n*, *m*), for *n* number of variables, *m* number of clauses.

SAT is NP-hard: **Cook's theorem**.

k-SAT problem: Number of literals in each clause in the SAT problem is limited to *k*, where *k* is obviously an integer.

It is not unreasonable to expect that *k-SAT*, for a constant integer *k*, to be possibly a P-class problem! K-SAT is restricted version of SAT.

2-SAT problem is indeed *P-class*: Davis-Putnam algorithm can find an assignment of variables in polynomial time, always (for all problem instances), when $k<=2$.

BUT,

3-SAT is NP-hard

The chosen source problem, which is already proved to be NP-hard, is the SAT problem itself. [Note, 3-SAT is an "easier" restricted version of the SAT problem.]

Transformation from **ANY** instance of SAT to a derived instance of 3-SAT is given below.

Classify SAT clauses into 4 groups: 1-clauses, 2-clauses, 3-clauses, and pclauses with p>3 literals in these clauses.
We will (1) show transformations (to the corresponding 3-clauses) for each of these 4 types of clauses;
(2) prove that the transformations are correct, i.e., truth preserving; and then
(3) we will discuss the polynomial nature of the aggregate-transformation at the end.

1-clauses:

Say, {u}, where u is a literal (not just a variable, but a variable or its negation).
The corresponding 3-clauses (for the target 3-SAT problem) will need two additional variables, say, z1, z2.
The derived 3-clauses are: {u, z1, z2}, {u, ~z1, z2}, {u, z1, ~z2}, {u, ~z1, ~z2}.
All these 3-clauses can be True only if *u* is True, and vice versa (hence this is a correct transformation).

2-caluses:

Say, {u1, u2}.

The corresponding 3-clauses (for the target 3-SAT problem) will need one additional variable, say, z.
The derived 3-clauses are: {u1, u2, z}, {u1, u2, ~z}.
Both the 3-clauses can be *True* only if either of u1 or u2 is *T* (i.e., the source clause in the source SAT problem instance), and vice versa. Hence this is a correct transformation.

3-caluses:

Say, {u1, u2, u3}.
Just copy them to the target problem instance, they are already 3-clauses.

p-caluses:

Say, $\{u_1, u_2, \ldots, u_{k-1}, u_k, u_{k+1}, \ldots, u_p\}$, where 1□k□p.
Create the following 3-clauses (for the target 3-SAT problem):
$\{u_1, u_2, z_1\}$, $\{\sim z_1, u_3, z_2\}$, $\{\sim z_2, u_4, z_3\}$, …, $\{\sim z_{k-3}, u_{k-1}, z_{k-2}\}$, $\{\sim z_{k-2}, u_k, z_{k-1}\}$, $\{\sim z_{k-1}, u_{k+1}, z_k\}$, …, $\{\sim z_{p-3}, u_{p-1}, u_p\}$.
This needs creation of additional (p-3) variables: $z_1, z_2, \ldots, z_{p-3}$.

In order for the *source (SAT) p-clause to be True*, at least one of the literals $u_1, \ldots, u_p$ has to be true. Without any loss of generality assume that the *True* literal to be u_k. We do not care if more literals are *True*, assume all the rest to be *False*.
Then the following assignment of new variables (*z*'s) (along with u_k=T) will make all the derived 3-clauses *True*: z_1=T, z_2=T, …., z_{k-3}=T**,** z_{k-2}=T, z_{k-1}=F, …, z_{p-3}=F. [CHECK IT] Hence, if the source *p*-clause has a *True* assignment, so do all the derived corresponding 3-clauses.
Note that the construction algorithm does not care about which literal is really *True*.

Assume the source p-clause to be False, i.e., none of the literals $u_1, \ldots, u_p$ is *True*. Try any set of assignments for the new variables (*z*'s), at least one clause will always remain *False*, you cannot make all of them *True* just by assigning the new variables.
Try one such assignment: Suppose, z_1=T (in order to make the first derived 3-clause *True*),

then z_2 should be = T, and then z_3=T, …., z_{p-3}=T (in order to make the lastbut-one derived 3-clause *True*), but that makes the LAST derived 3-clause is *False*.
Try any other combination of assignments, you will always run into the same problem – *at least one derived clause will become false*, because that is how the derived 3-clauses are constructed!

Hence, the transformation-scheme for the p-clauses is *truth-preserving*, or the algorithm is *correct*.

Complexity of the transformation algorithm:

Say, the number of source (SAT) 1-clauses is k_1, 2-clauses is k_2, 3-clauses is k_3, and p-clauses are k_p (depending on the values of p).

Then, the number of variables created in the target 3-SAT problem instance would be = $n + 2k_1 + k_2 + \square_p [(p-3)k_p]$.

The number of derived 3-clauses would be = $4k_1 + 2k_2 + k_3 + \square_p [(p-2)k_p]$.

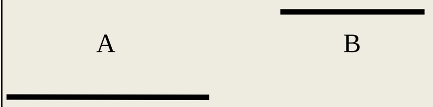
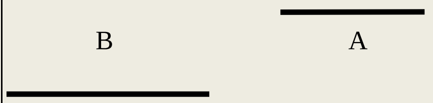
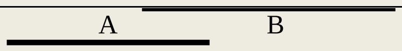
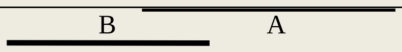
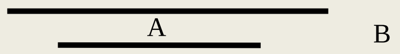
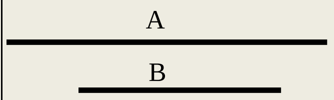
The total number of steps in creating the new variables and the new 3clauses are the sum of these two polynomials, hence a *polynomia*l with respect to the input problem size (#variables and #clauses in the source SAT problem instance).

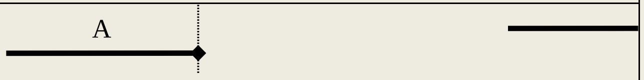
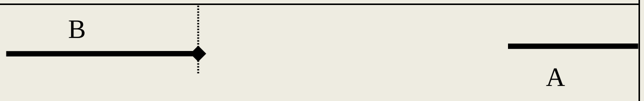
Thus, we have a <u>correct (truth preserving) & polynomial</u> problem transformation algorithm from any SAT problem instance to a corresponding 3-SAT problem instance => **3-SAT is NP-hard** (given the fact that SAT is NP-hard).

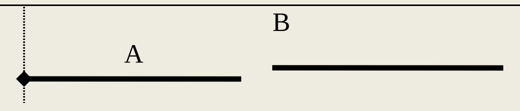
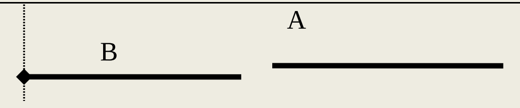
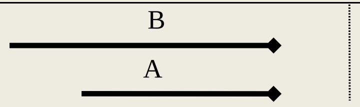
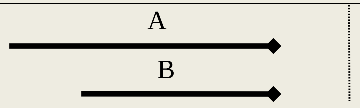
Since we have shown before that SAT is in NP-class, 3-SAT is obviously in NP-class. This concludes the proof that **3-SAT is NP-complete**.

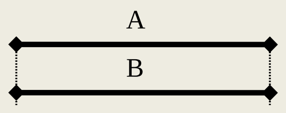
<u>*Interval Temporal Reasoning Problem is NP-hard*</u>

Basic Relations between Pairs of Intervals (B):

Binary operator	Relation type	Example	Representation
b	before	A(p)B	
~b	before inverse	A(~p)B	
o	overlap	A(o)B	
~o	overlap inverse	A(~o)B	
d	during	A(d)B	
~d	during inverse	A(~d)B	

Binary operator	Relation type	Example	Representation
m	meet	A(m)B	
~m	meet inverse	A(~m)B	

s	start	A(s)B	
~s	start inverse	A(~s)B	
f	finish	A(f)B	
~f	finish inverse	A(~f)B	

Binary operator	Relation type	Example	Representation
eq	equal	B(eq)A	

QTCN: A *qualitative temporal-constraint network* (QTCN) is a graph $G=(V, E)$, where each node is an interval, and each directed labeled edge (v_1 (R) v_2,)$\in E$ represents disjunctive constraint R from v_1 to $v_2 \in V$, where $R \subseteq 2^B$.

ITR Problem: Given a QTCN does there exist an assignment for each node on the time-line such that all the constraints are satisfied.

Example: QTCN[V = {i1, i2, i3}, E = {(i1 (p|o) i2), (i1 (m) i3), (i2 (o|a) i3)}]. Answer to ITR: Yes. Draw i1, i2 and i3 on timeline in such a way that i1 overlaps i2 and meets i3, while i2 overlaps i3. ITR is NP-hard: transform arbitrary 3-SAT problem to the corresponding ITR problem.

Let, the input 3-SAT problem is C = {(li1, li2, li3) | i = 1, …, m}, i indicates the clause, lik indicates k-th literal in the i-th clause.

Create an interval called *spl*. Create an interval Iij for each literal lij. Imagine left of *spl* will have the True literals (when the 3-SAT is assigned) and right of *spl* will contain the False literals. All three literals of a clause can be on the left of *spl* but not more than two can be on the right, when the input clause is satisfied by the assignment. This is the trick in creating interval relations.

First, fix the intervals so that they can be on either side of *spl* but not on both the side, with a constrained end point:
For all i=1,..,m and j=1, 2, 3 Iij
(m|~m) spl.

Then the other constraint, for each clause i = 1..m,
Ii1 (f | ~f | s | p | ~p) Ii2
Ii2 (f | ~f | s | p | ~p) Ii3
Ii3 (f | ~f | s | p | ~p) Ii1

Finally, we want to make sure that when two literals appear with opposite sign in different clauses, their corresponding intervals go on different sides of the interval *spl*.
For each lij = ~lgh, a constraint
Iij (p | ~p) Igh

This construction ensures that if the source 3-SAT is satisfiable one can easily satisfy the generated ITR. However, if the source 3-SAT is unsatisfiable (at least on clause, say, Ck is unsatisfiable by any given assignment), then the corresponding intervals in the ITR cannot be assigned as all the three intervals for that clause will try to be on the False side, which is impossible as per the construction ('s' relation's inverse is absent thus, creating a cycle over the intervals Ik1, Ik2, and Ik3).

<u>3D Matching is NP-complete</u>

2DM Problem: Two finite sets of individuals of equal cardinality: M = {m1, m2, … mk}, F = {f1, f2, …, fk}. A set of acceptable pairs of individuals from the two groups: A = {(fi, mj), 1 □ i,j □ k} = {(f3, m1), (f5, m1), (f3, m2), (f9, m2), …}.
Question: Does there exist a matching set of tuples A' within A (subset of A) such that every individual is in a tuple in A' once and only once. The cardinality of A' will be k, if it does exist.

3DM Problem: 3 sets (or "types") of individuals, X, Y, W, with same cardinality. Each element in A is a 3-tuple, A = ((x_i, y_j, w_l), 1 □ i, j, l □ k} from those three sets. *Question*: similar as above (does there exist a matching subset A' within A).

3DM is NP-class: Given a matching set A’ one has to check: (1) Every individual is covered or not; (2) Any individual is covered twice or not. It can be done in polynomial time :: check in a 3D coordinate system.

3DM is NP-hard: Polynomial transformation from 3-SAT to 3DM

Source 3-SAT Problem: Variables $U=\{u_1, u_2, \ldots, u_n\}$, 3-Clauses $C=\{c_1, c_2, \ldots, c_m\}$.

We have to construct 3 sets of individuals X, Y, and W; and a set of triplets A from them. *Truthfulness property needs to be preserved*. Transformation algorithm must be polynomial with respect to number of variables and clauses in the input 3-SAT problem.

3 types of triplets in A: "truth-setting and fan-out," "satisfaction testing" (of source 3-SAT), and "garbage collection" (to make sure everybody is covered in the target 3DM instance).

<u>Truth-setting + Fan-out components of A</u>:
In each clause each variable may appear only once (as Positive or as Negative)
For each variable u_i (1 □ i □ n) and each clause c_j (1 □ j □ m): two sets of triplets in A:

$T_{pos\_i} = \{ (\underline{u}_i[j], a_i[j], b_i[j]) : 1$ □ j □ $m\}$
$T_{neg\_i} = \{ (u_i[j], a_i[j+1], b_i[j]) : 1$ □ $j < m\} + \{ (u_i[m], a_i[1], b_i[m])\}$

a's are in X (m), and b's are in Y (m), u's and $\underline{u}$’s are in W (2m).

For every variable u_i, the matching will have to pick up exactly m triplets from A (out of 2m), for A’, depending on whether u_i is assigned *True* (T_{pos_i}) or *False* (T_{neg_i}). [FIGURE below from Garey and Johnson’s book]

Total number of triplets = 2mn

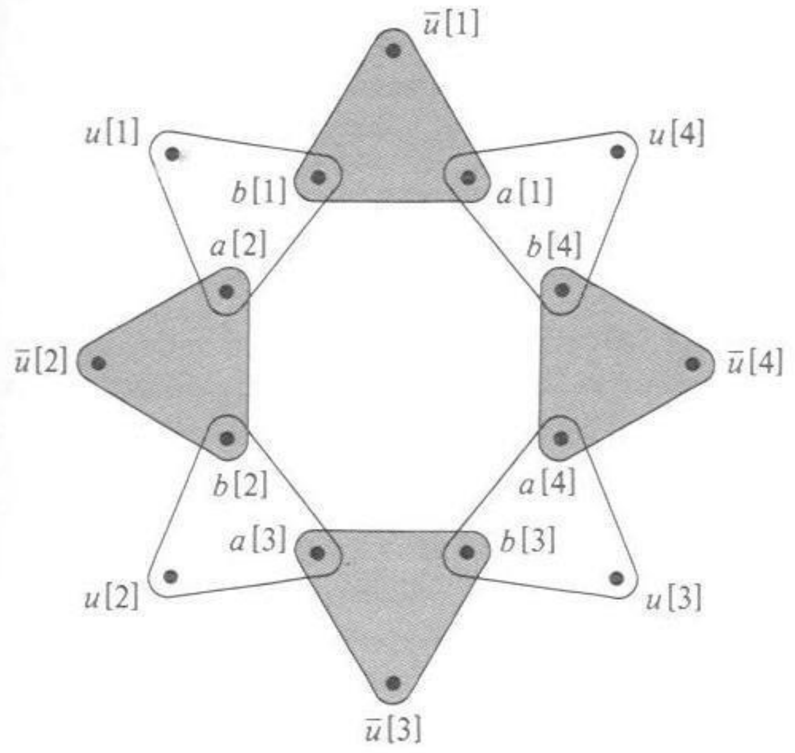


Figure 3.2 Truth setting component T_i when $m=4$ (subscripts have been deleted for simplicity). Either all the sets of T_i^t (the shaded sets) or all the sets of T_i^f (the unshaded sets) must be chosen, leaving uncovered all the $u_i[j]$ or all the $\bar{u}_i[j]$, respectively.

One such chain for each variable.

Satisfaction-testing components of A:
A triplet for each literal in each clause (**3m in number**), this is where we need 3-SAT (or a k-SAT, for a fixed integer k, as a source problem, as opposed to the general SAT problem - in the later case this number may not remain polynomial).

For each clause c_j three triplets (corresponding to three literals in it):
Cj = { either ($\underline{u_i}$[j], s_a[j], s_b[j]) if ~u_i is in c_j, or (u_i[j], s_a[j], s_b[j]) if u_i is in c_j}.
Cj's are components within A.

When the input/source 3-SAT has a satisfying assignment, the variable u (or $\underline{u}$) corresponding to the *True* literals will not have match as per previous construction (with a's and b's), and hence this construction Cj will get them matched with corresponding s[]'s. On the other hand if a clause has all its literals *False*, every u (or $\underline{u}$) in the corresponding Cj is matched already in the previous construction, but if you do not pick up from Cj, then the corresponding s's remain unmatched.

Garbage collecting components of A:

To get those individuals who are still not matched in A: not all variables occur in every clause, but the corresponding u's are created anyway

Two sets of triplets in A for each variable u_i and each clause cj:

G = {($\underline{u}_i$[j], g_a[k], g_b[k]), (u_i[j], g_a[k], g_b[k]) : 1 <= k <= m(n-1), 1 <= i <= n, 1 <= j <= m}.

One set per k, k runs up to m(n-1). **Total triplets (2mn).m(n-1).** For the u's (or $\underline{u}$'s) that are not in A' (or, not in A???).

Sets of individuals:

W = {$\underline{u}_i$[j], u_i[j] : 1□i□n, 1□j□m},	**2mn** in number
X = Aa + S1 + G1	
Aa = {a_i[j] : 1□i□n, 1□j□m},	mn
Sa = {s_a[j] : 1□j□m},	m
Ga = {g_a[k] : 1□k□m(n-1)},	m(n-1)
	Total=**2mn**
Y = Bb + S2 + G2	
Bb = {b_i[j] : 1□i□n, 1□j□m},	mn
Sb = {s_b[j] : 1□j□m},	m
Gb = {g_b[k] : 1□k□m(n-1)},	m(n-1)
	Total=**2mn**

Polynomial construction:

#individuals + #triplets = 6mn + (2mn + 3m + 2(m^2)n(n-1))

Correctness:
If C is not satisfiable A' cannot be constructed. If A' is constructed, then the triplets in A' corresponding to the s1's and s2's are the truth-setting components, make the corresponding literals $\underline{u}$ or u's from those triplets *True*, there is no way the same variable will get multiple conflicting assignments by this.

If C is satisfiable:
Say, there exists an assignment for each u_i (*True*, or *False*) which makes C *True*.
For constructing A':
Choose a literal from each clause that is *True* (one must exist), and find corresponding triplet from the Satisfaction-testing components, for marrying each corresponding s1 and s2.
Depending on if u_i is *True* or *False* choose the set T_{pos_i} or T_{neg_i}, to marry off a's and b's.
Pick up appropriate triplets from G to take care of g_a's and g_b's, for those u_1's and u_2's who are still unmarried.
Thus, A' can be constructed, from the source 3-SAT's variable assignments.

An Example with SAT->3DM trans:

3-SAT: var: (u1, u2, u3), clause: {(u1, ~u2, u3)}

3DM:

Truth setting construction:
A triplets: Unmatched individuals:
Tpos1=(~u11, a11, b11) u11, Tneg1=(u11, a11, b11) W={~u11, ~u21, u21, ~u31, u31}
Tpos2=(~u21, a21, b21) a21, Tneg2=(u21, a21, b21) X={a11, a31}
Tpos3=(~u31, a31, b31) Tneg3=(u31, a31, b31) Y={b11, b21, b31}
A' match set for sat assignment (say, for u1=T, u2=T and u3=F) in source-3SAT problem:

{(~u11, a11, b11), (~u21, a21, b21), (u31, a31, b31)}

Unmatched individuals left after this:

W={u11, u21, ~u31}, X={none}, Y={none}

Satisfaction testing construction:

A triplets: Unmatched individuals:

C1={(u11, sa1, sb1), (~u21, sa1, sb1), (u31, sa1, sb1)} W={u11, u21, ~u31}

X={sa1, sa2}, Y={sb1, sb2}

A’ match set for truth setting of the clause, say, by u1=T:

{(u11, sa1, sb1)}

Unmatched individuals left after this:

W={u21, ~u31}, X={none}, Y={none}

Garbage collection/ create new individuals to match all:

A triplets: Unmatched individuals:

G={ (~u11, ga1, gb1), (u11, ga1, gb1), W={u21, ~u31}

(~u21, ga1, gb1), (u21, ga1, gb1), ga2} X={ga1,

(~u31, ga1, gb1), (u31, ga1, gb1), gb2} Y={gb1,

(~u11, ga2, gb2), (u11, ga2, gb2),

(~u21, ga2, gb2), (u21, ga2, gb2),

(~u31, ga2, gb2), (u31, ga2, gb2)}

1<=k<=1(3-1), 1<=i<=3, 1<=j<=1;

A’ match set for truth setting of the clause, say, by u1=T:

{(u21, ga1, gb1), (~u31, ga2, gb2)}

Unmatched individuals left after this:

W={None}, X={none}, Y={none}

End example.

You may like to try with a unsatisfiable source 3-SAT:
{u, v, w}, {u, v, ~w},{u, ~v, w},{u, ~v, ~w}, {~u, v, w}, {~u, v, ~w},{~u, ~v, w},{~u, ~v, ~w}.
Take u=v=w=T that makes the last clause *False*. Try to create triplets in A for that clause and see why you cannot create match for corresponding sa's and sb's created in the Truth-setting component of the construction.

Reasoning with Cardinal-directions algebra

Problem definition: Input: A set of points V, and a set E of binary relations between some pairs of points in V, v_i R_{ij} v_j where
R_{ij} is a disjunctive subset of the set of nine basic relations {Eq, East, …} in Cardinal directions-calculus (Figure below).
Question: Can the points in V be located in a real space of two-dimensions.

Example input (1): V={v1, v2, v3, v4}, and E = {(v2 (Northeast, North) v1), (v3 (Northeast) v2), (v3 (West, Southwest, South) v1), (v4 (North) v3)}.
Answer: No satisfying placement of v1, v2, v3 and v4 exists following these constraints.
Example input (2): V={v1, v2, v3, v4}, and E = {(v2 (Northeast, North) v1), (v3 (Northeast) v2), (v3 (Northeast) v1), (v4 (North) v3)}. Answer: Yes. v1 = (0,0), v2=(1,1), v3=(3,3), and v4=(3,4) is such a satisfying assignment.

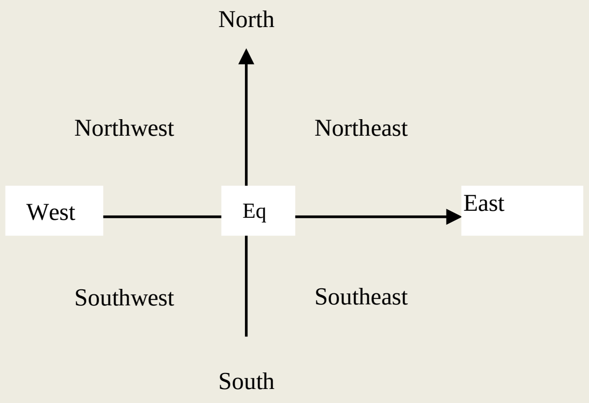


Figure 1: 2D-Cardinal directions calculus

Theorem: reasoning with 2D-Cardinal directions algebra is NP-hard.

Proof (Ligozat, Jnl. of Visual Languages and Constraints, Vol 9, 1998):

Proof by constructing a Cardinal-directions algebra problem instance from an arbitrary 3-SAT problem instance with a set of clauses like $C_i = \{ l_{i,1} \mid l_{i,2} \mid l_{i,3} \}$.
(1) For every literal l_{ij} create two points P_{ij} and R_{ij} such that P_{ij} {nw, n, ne, e, se} R_{ij}, and (2) for every clause C_i we have P_{i1} {sw, w, nw} R_{i2} and P_{i2} {sw, w, nw} R_{i3} and P_{i3} {sw, w, nw} R_{i1}. Also, (3) for every literal l_{ij} that has a complementary literal l_{gh} we have two relations between their corresponding points: P_{ij} {sw, s, se} R_{gh} and P_{gh} {sw, s, se} $R_{ij.}$

Writing P as (p, q) and R as (r, s) we get three relations on x-axis and y-axis,
(1) $(p_{ij} > r_{ij}) \mid (q_{ij} > s_{ij})$,
(2) $(p_{i1} < r_{i3})$ & $(p_{i2} < r_{i1})$ & $(p_{i3} < r_{i2})$,
(3) If l_{ij} and l_{gh} are complementary literals, $(q_{gh} < s_{ij})$ & $(q_{ij} < s_{gh})$

We express $(p_{ij} > r_{ij})$ as false whenever (if and only if) any literal l_{ij} is *true* in clause C_i.
So, for the six points corresponding to any unsatisfiable clause where all literals are *false*, generates three relations for their x-coordinates, $(p_{i1} > r_{i1})$ & $(p_{i2} > r_{i2})$ & $(p_{i3} > r_{i3})$ by construction (1). In addition by construction (2) we get three more relations $(p_{i1} < r_{i3})$ & $(p_{i2} < r_{i1})$ & $(p_{i3} < r_{i2})$. You cannot assign these six points on the x-axis satisfying all these six relations. Hence, if any clause is *false* (by any truth assignment of literals), then the corresponding Cardinal-directions algebra problem does not have a solution (because these six points cannot be assigned in space).

On the other hand suppose there exists a truth assignment that satisfies all clauses. So for every clause C_i there is at least one literal l_{ij} *true*, or $(p_{ij} > r_{ij})$ is false, or $\mathbf{(q_{ij} > s_{ij})}$ is true (by construction (1)). Then, a complementary literal l_{gh} in clause C_g will create the following relations: $\mathbf{(q_{gh} < s_{ij})}$ **&** $\mathbf{(q_{ij} < s_{gh})}$**.** The three relations are not consistent with $\mathbf{(q_{gh} > s_{gh})}$ that should be true if and only if l_{gh} is also *true*. Hence, l_{ij} and l_{gh} cannot be *true* at the same time, thus, prohibiting any inconsistency for the truth assignment of literals across the clauses. In other words, the cross-linking of the points across the corresponding literals will not prohibit from their being put in the space. Note that as soon as a literal is *true* in a clause the corresponding six points can be assigned as per constructions 1 and 2, only construction 3 might have created problem via the complementary literals.

Above argument proves the correctness of the constructions.

The number of steps in the construction algorithm involves: constructing 6 points per clause, 6 relations per clause from construction (1), 3 relations per clause from (2), and at the most 3 relations per pair of clauses. Total number is polynomial with respect to the numbers of variables and clauses. Hence, the above construction is a polynomial transformation from 3-SAT problem to the 2D-Cardinal algebra problem.
QED.

OTHER MODELS OF COMPUTATION

NP-hardness came as a big shock to the progress of computing. Many of the important and interesting problems turn out to be NP-hard!

Areas where problems are not NP-hard see success in the market place, e.g., most problems in data-management or linear programming. Other areas where problems are NP-hard either bypass (see discussion above on how to tackle NP-hard problems) or simply continue to exist as a research topic for a long time (e.g., artificial intelligence, or software engineering).

Note that the model of computation in which some problems are NP-hard is the Turing/Church (TM) model of computation.

The first hope was raised with **parallel computing**. Alas, that turned out to be not more powerful than the present model! Significance: even if infinite number of CPU's are provided and communication time is 0 between them, the NP-hard problems may not still have any distributed poly-algorithms.

Artificial Neural Network provides a different model of computation. It has the same power as the Turing Machine, but it solves some pattern recognition problems very fast.

An esoteric approach is the **DNA computing**. They solve some graphtheoretic problems [e.g. TSP] in a very small number of steps (but with a huge time per step with the current bio-technology). But DNA computing is proved to be as powerful as Turing Machine (i.e., NP-h remains NP-h even with DNA computing). Researchers are trying to develop computers out of DNA's.

Another limitation to the conventional computing is surfacing from the *hardware*: circuit densities on chips are becoming too large: avoiding crossfeeds between circuit-elements will become impossible soon, because of the
"quantum-tunneling" effects.

A new technology shaping up on the horizon is in the form of **Quantum computing**. Multiple data (and instructions) can be "superimposed" in the same quantum-register. When needed, a quantum operator will "extract" the required output. An algorithm becomes a Quantum operator in this paradigm.

Physical technological hurdles are being overcome gradually. It seems we have been doing quantum computing unknowingly for the last few decades in the form of NMR (nuclear magnetic resonance, as in MRI in medicine), where atomic nuclei provide the Quantum bits (Q-bits).

Quantum computing (QC) is a different model than TM, has in-built nondeterminism. But, apparently NP-hardness remain NP-hard their too *(find out a relevant paper for a free lunch from me)!* However, speed, space complexity, and power consumption of QC may be much lower than those of the electronic computers.

Algorithms for QC are quite different than those for conventional computers (based on Turing machine). QC may also provide a better model for security, which attracts cryptographers and the communications industry.
QC has also generated topic called *Spintronics* where electrons' spin is used instead of charge for building digital circuitry (see IBM research).