CS 3100, Models of Computation, Spring 20, L24

Ganesh Gopalakrishnan
School of Computing
University of Utah
Salt Lake City, UT 84112

URL: bit.ly/3100s20Syllabus

Computability Review to do Asg-7

- You can do Questions 1 and 2 rightaway!
- In this lecture, we will introduce the idea of mapping reductions through short examples
- After seeing enough of these, you will know what is being attempted (once you get this, you can begin constructing your own MRs)
- In the following, TMs will be presented in familiar C-style syntax, using function names M, M', M1, etc
- All TMs will be presented as if they take a single input argument "x" . Sometimes, we will denote inputs by "w". You'll see how the usage goes. Most often, "w" is the input provided for the mapping-reduction process.
- We will show their accept/reject states via "labels" of the form Accept_M and Reject_M. Sometimes, these label(s) may be omitted – but assume they exist always.
- We shall introduce functions Translate, Translate', Translate1, etc. that produce TMs in terms of other TMs
- Note: I've placed "final batteries". They help review for MT-3 also.

What is the language of this TM?

```
M ( x ) {
Goto Accept_M;
Accept_M: // stuck
}
```

What is the language of this TM?

```
M ( x ) {
Goto Accept_M;
Accept_M: // stuck
}
```

Ans: Sigma*

What is the language of this TM?

```
M ( x ) {
Goto Reject_M;
Reject_M: // stuck
}
```

What is the language of this TM?

```
M ( x ) {
Goto Reject_M;
Reject_M: // stuck
}
```

Ans: { }

What is the language of this TM?

```
M ( x ) {
Loop; // infinite loops are denoted by the “Loop” command
Accept_M: // stuck
}
```

What is the language of this TM?

M (x) {
Loop; // infinite loops are denoted by the “Loop” command
Accept_M: // stuck
}

Ans: { }

What is the language of this TM?

```
M ( x ) {
If x = 101 goto Accept_M;
Loop;
Accept_M: // stuck
}
```

What is the language of this TM?

```
M ( x ) {
If x = 101 goto Accept_M;
Loop;
Accept_M: // stuck
}
```

Ans: { 101 }

What is the language of this TM?

```
M ( x ) {
If x = 101 goto Accept_M;
    Accept_M: // stuck
Else
   Accept1_M: // stuck
}
```

What is the language of this TM?

```
M ( x ) {
If x = 101 goto Accept_M;
    Accept_M: // stuck
Else
   Accept1_M: // stuck
}
```

Ans: {101} Union { all others} i.e. Sigma*

What is the language of TM M' ?

```
TM M = …a given TM …   // think of this as a global constant
Input w = …a given w… // think of this as a global constant
// Now define a new machine M' in terms of M and w

M' ( x ) {
If x = "101" goto Accept_M;
Run M on w;
// Phew, got here!
Accept_M': // stuck
}
```

What is the language of TM M' ?

```
TM M = …a given TM …   // think of this as a global constant
Input w = …a given w… // think of this as a global constant
// Now define a new machine M' in terms of M and w

M' ( x ) {
If x = "101" goto Accept_M;
Run M on w;
// Phew, got here!
Accept_M': // stuck
}
```

Ans:

- M' has language {101} if M does not halt on w
- M' has language Sigma* if M halts on w

What is the language of TM M' ?

TM M = …a given TM … // think of this as a global constant
Input w = …a given w… // think of this as a global constant
// Now define a new machine M' in terms of M and w

M' (x) {
Run M on w;
// Phew, got here!
If x = "101" goto Reject_M';
Accept_M': // stuck
Reject_M': // stuck
}

What is the language of TM M' ?

TM M = …a given TM … // think of this as a global constant
Input w = …a given w… // think of this as a global constant
// Now define a new machine M' in terms of M and w

M' (x) {
Run M on w;
// Phew, got here!
If x = "101" goto Reject_M';
Accept_M': // stuck
Reject_M': // stuck
}

Ans:

- M' has language { } if M does not halt on w
- M' has language Sigma* - {101} otherwise

What is the language of the TM M’ produced ?

```
Translate(M, w) {
Print(
“M’ ( x ) {
Run %s1 on %s2;
// Phew, got here!
If x = “101” goto Reject_M’;
Accept_M’: // stuck
Reject_M’: // stuck
}”, M, w);
}
```

What is the language of the TM M’ produced ?

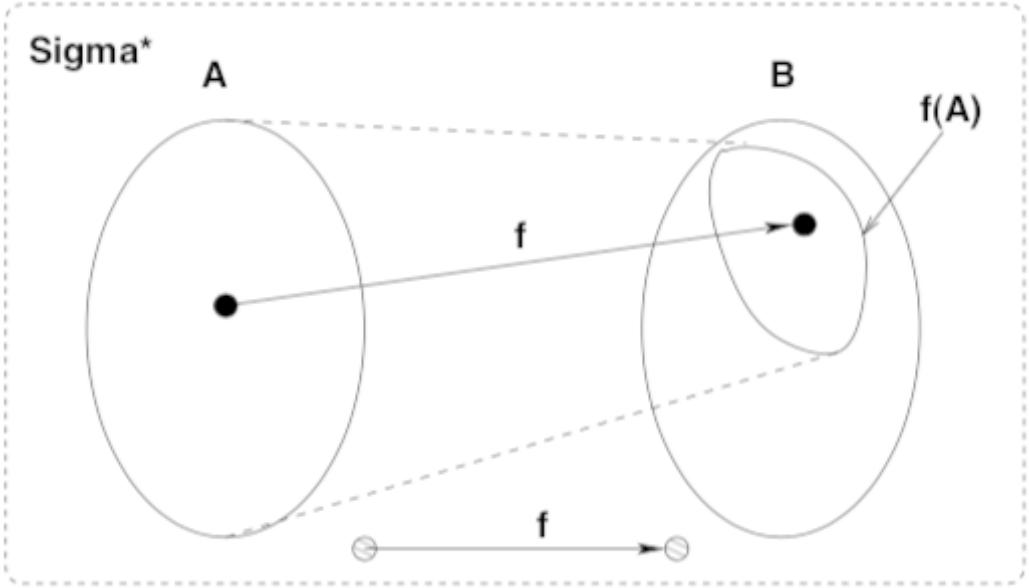
```
Translate(M, w) {
Print(
“M’ ( x ) {
Run %s1 on %s2;
// Phew, got here!
If x = “101” goto Reject_M’;
Accept_M’: // stuck
Reject_M’: // stuck
}”, M, w);
}
```

Ans:

- M’ has language { } if M does not halt on w
- M’ has language Sigma* - {101} otherwise

Why even introduce the “Translate” function?

- Function “Translate” is our Mapping Reduction function desired!
  - The MR is from language A to language B
- It must ensure that x in A if and only if f(x) in B; that is
  - x in A => f(x) in B
  - x not in A => f(x) not in B



Why do we build such a Translate function?

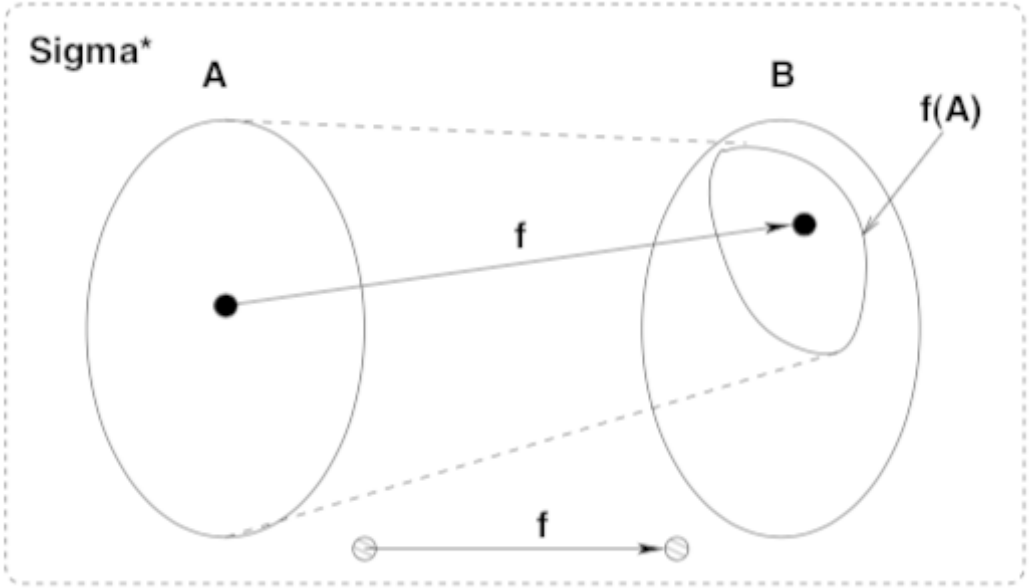
- We take “A” to be an ”already shown-impossible” problem
  - ”Shown-impossible” here means “shown to be undecidable”
- We are given a new problem “B”
- We want to (somehow) show that B is also impossible
  - I.e. show that B is also undecidable
- We try our best to build Translate such that
  - x in A => Translate(x) in B
  - x not in A => Translate(x) not in B
- If we succeed, then we have this going on:
  - Suppose someone claims to have a decider for B
  - We can then “invite” someone to give an “x” that falls into A
  - Then send Translate(x) to B’s decider, and take its decision as a decision for x in A
  - This violates the premis that A has a decider
  - Hence a decider for B cannot exist!

What all can the set “B” be?

- In general, set “B” is ANY INTERESTING QUESTION WE CAN ASK OF A TM !!
- The real result (known as Rice’s Theorem) says “no nontrivial property of a TM M’s language can be decided based on the syntactic description of M”
- For us, candidate “B” sets are
  - Given a TM M, does it have 101 in its language?
  - Given a TM M, does it have a regular language?
- Let’s build the Translate function for these two examples!

Specific example of “Translate” (or the ‘f’ fn.)

- Function “Translate” is our Mapping Reduction function desired!
  - The MR is from language A_TM to language Reg_TM
- It must ensure that x in A if and only if f(x) in B; that is
  - x in A => f(x) in B
  - x not in A => f(x) not in B



Translate function for “TM has 101 in its lang.”

```
Translate(M, w) {
Print(




);
}
```

Translate function for “TM has 101 in its lang.”

```
Translate(M, w) {
Print(
“M’ ( x ) {
Run %s1 on %s2;
If %s1 does not accept %s2, then loop;
// Got here because %s1 accepts %s2
Accept_M’: // stuck

}”, M, w);
}
```

Translate function for “TM has 101 in its lang.”

```
Translate(M, w) {
Print(
“M’ ( x ) {
Run %s1 on %s2;
If %s1 does not accept %s2, then loop;
// Got here because %s1 accepts %s2
Accept_M’: // stuck

}”, M, w);
}
```

- If M does not accept w, M’s language does not have 101 -- in fact it is empty
- If M does accept w, then M’s language does have 101 – in fact, it has EVERYTHING
- Thus, testing M’ having 101 or not leaks information on whether M accepts w. This provides a decider for A_TM

Translate function for “TM has a regular lang.”

```
Translate(M, w) {
Print(




);

}
```

Strategy:
Force M’ to have a regular language exactly when M accepts (or does not accept) w.

Translate function for “TM has a regular lang.”

```
Translate(M, w) {
Print(“
  M’(x) {

    Run %s1 on %s2 ;  // this is a different construction than taught last class!  Many constructions work !!
    // Phew reached here; so %s1 did not loop on %s2
    If %s1 accepted %s2 then if x is of the form 0^n 1^n then goto Accept_M1’ ;
    goto Reject_M1’;



}”, M, w
);
}
```

If M accepts w, M’ has a CFL i.e. { }

If M does not accept w, M’ has a regular language i.e. { }

NP-Completeness

Why NPC matters (from book by Garey/Johnson)

"I can't find an efficient algorithm, but neither can all these famous people."

Agenda: Chapters 16 and 17

- Before Chapter 16, we studied only Computability
  - Can a problem be solved algorithmically at all?
- We never asked “how long does such an algorithm take?”
- Complexity – Theory of NP-Completeness – will now be studied
  - Motivation: Can we solve it in Polynomial-time?
  - If the answer is not known, can we “link up all those important problems in some way such that solving any one of them using a P-time algorithm allows you to solve ALL of them using a P-time algorithm”

Chapters 16 and 17: Motivations for NPC

- Many problems of importance appear to have only exponential algorithms
- There is no proof (yet) that they do not have polynomial algorithms
- But in many cases they have Nondeterministic Polynomial-time algorithms (an NDTM can consume a poly amount of “fuel” and stop with a solution)
- Thus scientists grouped together problems into a class called NPC -- “NP-Complete”
- NPC are the hardest of NP problems
  - Any NP problem <=p an NPC problem
- Thus by solving a single NPC problem using a P-time algorithm allows them to solve ALL of NP problems using a P-time algorithm

Agenda: Chapters 16 and 17

- The first NP-Complete Problem studied was 3-SAT
  - 3-CNF Boolean Satisfiability

- Thus we will review Boolean Logic first (and then study NPC)

Review of Boolean Logic

Boolean variable

Boolean variables are syntactically denoted by variable names such as

x, y, z, etc.

Boolean literals

Literals are syntactically denoted by

x, !x, y, !z, etc.

Literals are variables or their negations

A clause

Clauses are syntactically written as

(x + !y + z)

They are disjunctions of literals

A 3-CNF clause is one that has 3 literals

3-CNF clauses are important in our discourse

A CNF formula

CNF formulae are syntactically conjunctions of clauses, written

(x + !y + z) . (!x + y + p) . (q + r + s) (the “dot” could be replaced by & or /\ )

or

(x + !y + z) (!x + y + p) (q + r + s) (the conjunctions could also be left out)

They are in Conjunctive Normal Form (CNF)

That means, a product (“and”) of clauses

A product term, and a DNF formula

In contrast, a DNF formula is a sum of products

A DNF formula is of the form

a . !b . c + !a . d . !e

DNF formulae won't have much of a role in the study of NPC, as they do not encode the hardness of NPC problems

(Of course one can convert DNF formulae to CNF

But that can blow-up the formula to an exponential size)

Satisfiability

CNF formulae are satisfied by satisfying every clause

(x + !y + z) . (!x + y + p) . (q + r + s)

Satisfiability

DNF formulae are satisfied by satisfying any product term

```
x . !y . z  +  !x . x . W  +  p . !x . z
```

Overall, 3CNF SAT is THE central problem

Have clauses, each containing 3 literals in a CNF formula

(a + b + !b) . (!a + a + d) . (a + e + !f)

Some 3CNF formulae are UNSAT (unsatisfiable). This one is:

(a + b + c) . (a + b + !c) . (a + !b + c) . (a + !b + !c) .

(!a + b + c) . (!a + b + !c) . (!a + !b + c) . (!a + !b + !c

Such formulae behave like a whack-a-mole

Whichever assignment you try, some clauses

acts as a “blocking clause” (a “spoiler”)

3SAT or 3CNF-SAT as a language

- 3SAT = { fmla : fmla is a 3CNF formula that is satisfiable }

Where we are going with this:

We will present a mapping reduction from 3-SAT to Clique

This helps us review the basics … and drive home NP-Completeness ideas

The language K-Clique

- Given an undirected graph, is there a set of K nodes such that they are all pairwise connected?
  - They form a K-Clique?

K-Clique – { < G > : G is a graph that has a K-Clique in it }

We are about to show 3-SAT <=p K-Clique

- <=p is a mapping reduction but with a polynomial bound on runtime
- That is, we can translate a 3-SAT instance to a K-Clique instance in polynomial time
- This means that if K-Clique has a Polynomial Algorithm, then 3-SAT will also have a Polynomial Algorithm

3SAT <=p K-Clique (the "Translate" fn.)

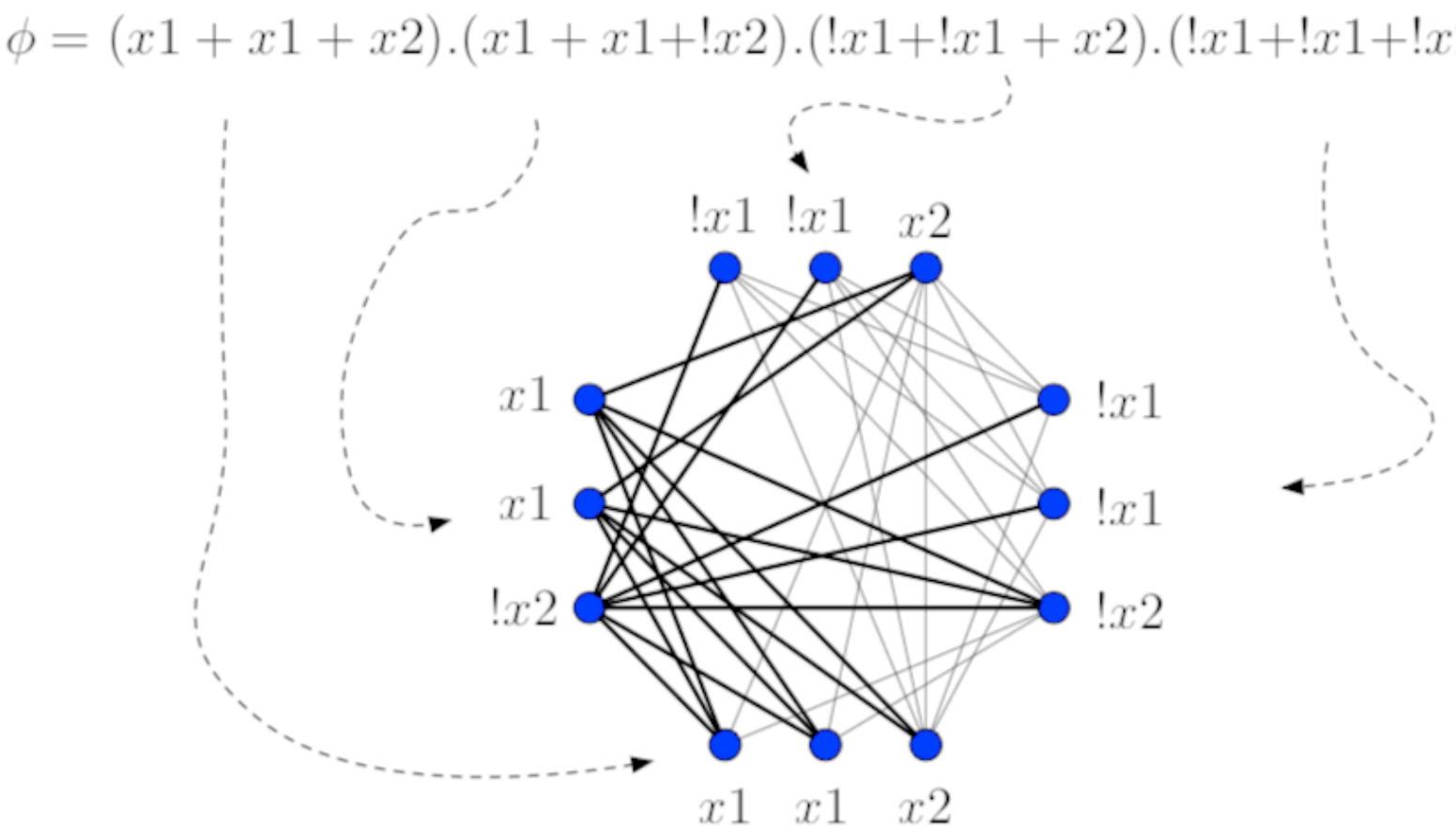


Figure 16.9: The Proof that *Clique* is NPH using an example formula $\varphi = (x1 + x1 + x2).(x1 + x1 + !x2).(!x1 + !x1 + x2).(!x1 + !x1 + !x2)$. We never connect the nodes within each clause "island" (there are four such islands, each with three nodes). Across each clause island, we draw edges in all possible ways *provided* we never connect a literal and its complement. For visual clarity, we show through dark edges all the edges emanating from the clause island for (x1+x1+!x2) going to all other clause islands. We also show the remaining edges, but using fainter lines.

Exercises to help learn 3-SAT <=p K-Clique

Practice Problems: Sat? Sat Instance? Clique?

$$\overset{=3}{(}x_1 + x_2 + x_3) \cdot (x_1 + !x_2 + !x_3)$$

Practice Problems: Sat? Sat instance? Clique?

$$\overset{=3}{(}x_1 + x_2 + x_3) \cdot (x_1 + !x_2 + !x_3)$$

Sat? Sat instance? Clique?

$$\overset{=3}{(}x_1 + x_2 + x_3) \cdot (!x_1 + !x_2 + x_3) \cdot (!x_1 + !x_2 + !x_3) \cdot (!x_1 + x_2 + !x_3)$$

Sat? Sat instance? Clique?

$$=3$$

$$(x_1 + x_2 + x_2) \cdot (!x_1 + !x_2 + !x_2) \cdot (x_1 + !x_2 + !x_2) \cdot (!x_1 + x_2 + x_2)$$

Your Asg-7's problems

- They help you practice these notions using the Binary Decision Diagram tool (BDD tool)
  - Part of Jove
- Binary Decision Diagrams will be explained now
  - They also are minimal DFA " in disguise "
- We will then study the theory of NPC armed with this knowledge

Binary Decision Diagrams

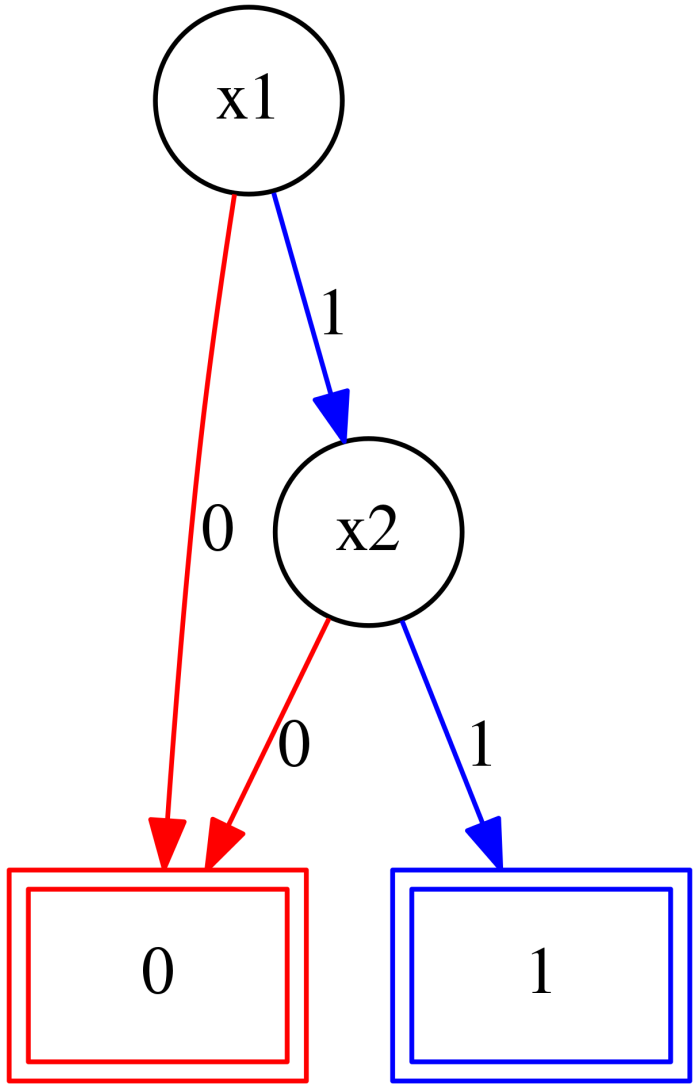
BDDs

- They are a data structure for representing Boolean Functions
- Included in Jove (see First_Jove_Tutorial/CH17/CH17.ipynb)

- All the details of BDDs is not that important
- But practice with our BDD tool will fix ideas in your mind better (seeing Boolean formulae as graphs is often edifying)

Use of BDD tool

```
# http://formal.cs.utah.edu:8080/pbl/BDD.php
# Very simple example to show-off syntax

#First declare the variables and specify variable orderin
Var_Order : x1 x2 x3

#Then define formula
fmla = (x1 | x2) & (x1 | !x2) & (!x1 | x2)

Main_Exp :  fmla

# Type “build BDD”

# Right-click and save PNG

# SAT, UNSAT, Valid – from shape of BDD
# SAT : paths exist to “1”
# UNSAT: [0] BDD
# Valid: [1] BDD
```



Things to observe about BDDs

- They are DAGs with one “layer” per variable
- They “decode” the formula in a top-to-bottom order
- The order of decoding determines the BDD size
- There are often good heuristics to select this order

Practice Problems: Sat? BDD for var order x1,x2,x3 ?

$$=^{3}(x_1 + x_2 + x_3) \cdot (x_1 + !x_2 + !x_3)$$

Sat? Sat instance with BDD var order x1,x2,x3?

$$\overset{=3}{(}x_1 + x_2 + x_3) \cdot (!x_1 + !x_2 + x_3) \cdot (!x_1 + !x_2 + !x_3) \cdot (!x_1 + x_2 + !x_3)$$

TMs help precisely model time complexity

- Here, we need to consider TMs solving problems involving Boolean expressions

Show relative hardness of problems via <=p

People observed that many real-world problems are expensive to solve in the worst case

E.g. Is there a clique in a large graph

The internet is a huge graph

Update of internet node software, reliable direct links between cities, … are all hard problems

They are all formally connected via <=p

(poly-time mapping reductions)

NPC problems are easy to check; difficult to solve

Both for 3SAT

(a + b + !b) . (!a + a + d) . (a + e + !f)

And clique

Definition of P-time and NP-time (from book)

- P-time: An algo is P-time if its computational tree is bounded in height by a polynomial function of the length of its input for every input in the language that the DTM decides. For this simple "101" DTM, here is that DTM's code and here is a computational tree – with paths shown. The paths are two for rejecting runs and one for an accepting run.

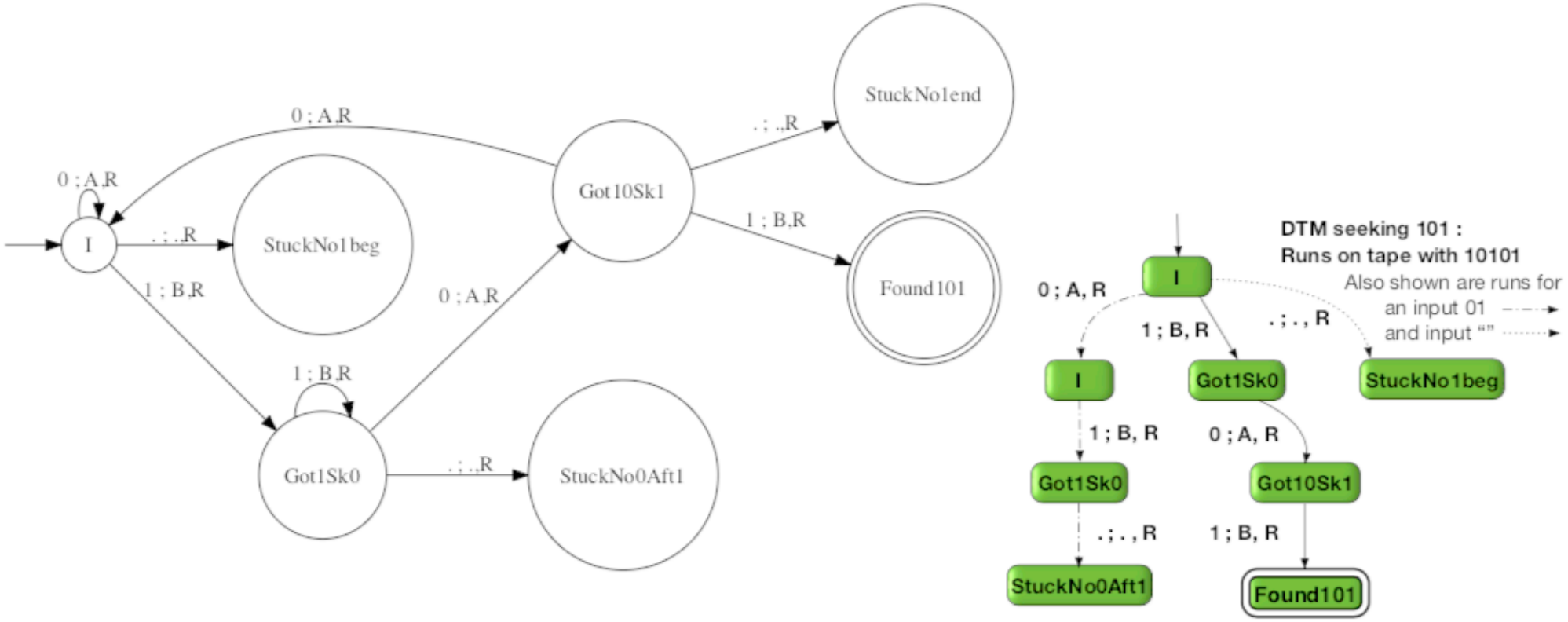
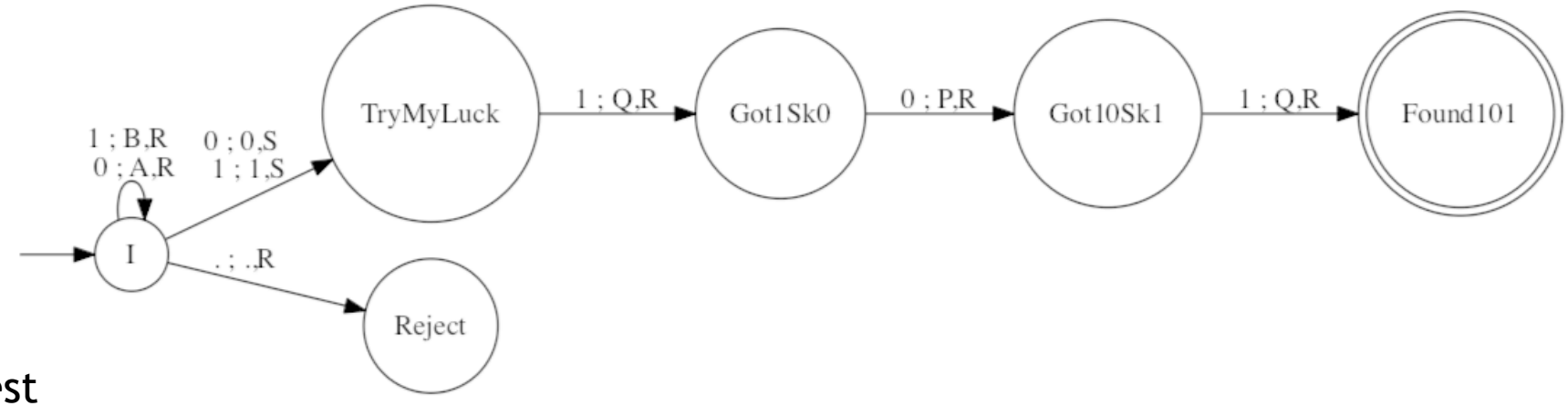


Figure 16.2: Transition diagram and computation tree for a DTM that looks for 101 within given w.

Definition of NP-time (from book)

- NP-time: An algo is NP-time if an NDTM can be obtained where it can guess a solution nondeterministically but be able to CHECK that solution in P-time. So the depth of the worst-case (deepest) path in that NDTM must be P-bounded for any input. Some problems may not even qualify for the “check phase” being P-bounded… but many useful problems have !! That makes the theory of NPC interesting and relevant in practice!

Illustration of NP-time (from book)



Look for the deepest path which accepts. That corresponds to The NP-time.

In Jove, the Fuel Models this depth Faithfully even for NDTMs…

(modulo bug-fixes if any…)

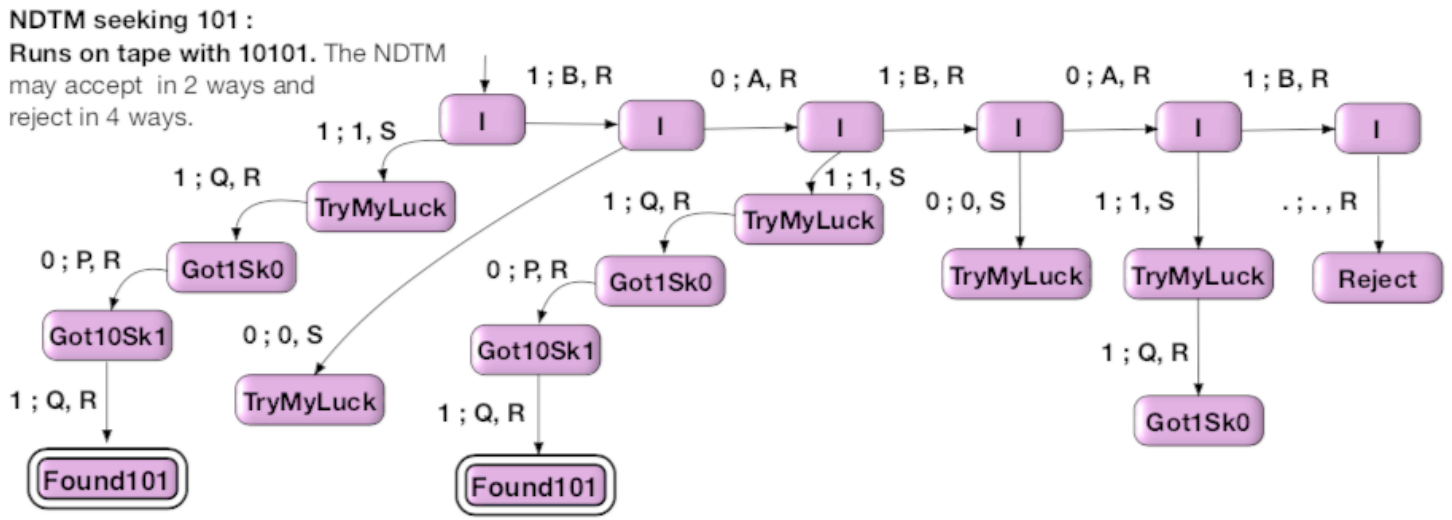


Figure 16.3: Transition diagram and computation tree for an NDTM that looks for 101 within given w.

Definition of P-time and NP-time (from book)

- P-time: Illustrated wrt 3SAT
  - Obtain computation tree of DTM for the language 3SAT (all members in it)
  - Can we claim anything about the depth of the computational tree for all inputs?
    - As far as we know, any DTM working on 3SAT appears to incur an exp depth for at least some of the instances
- NP-time: there is an NDTM that can guess the solution for a 3SAT problem (in P-time) – and also check this guess in P-time
  - Question: will we ever get a DTM that does this in P-time??
  - This is what the question of P =?= NP really means

Smart idea

- Define the idea of the hardest problems in NP
- Call it NPC

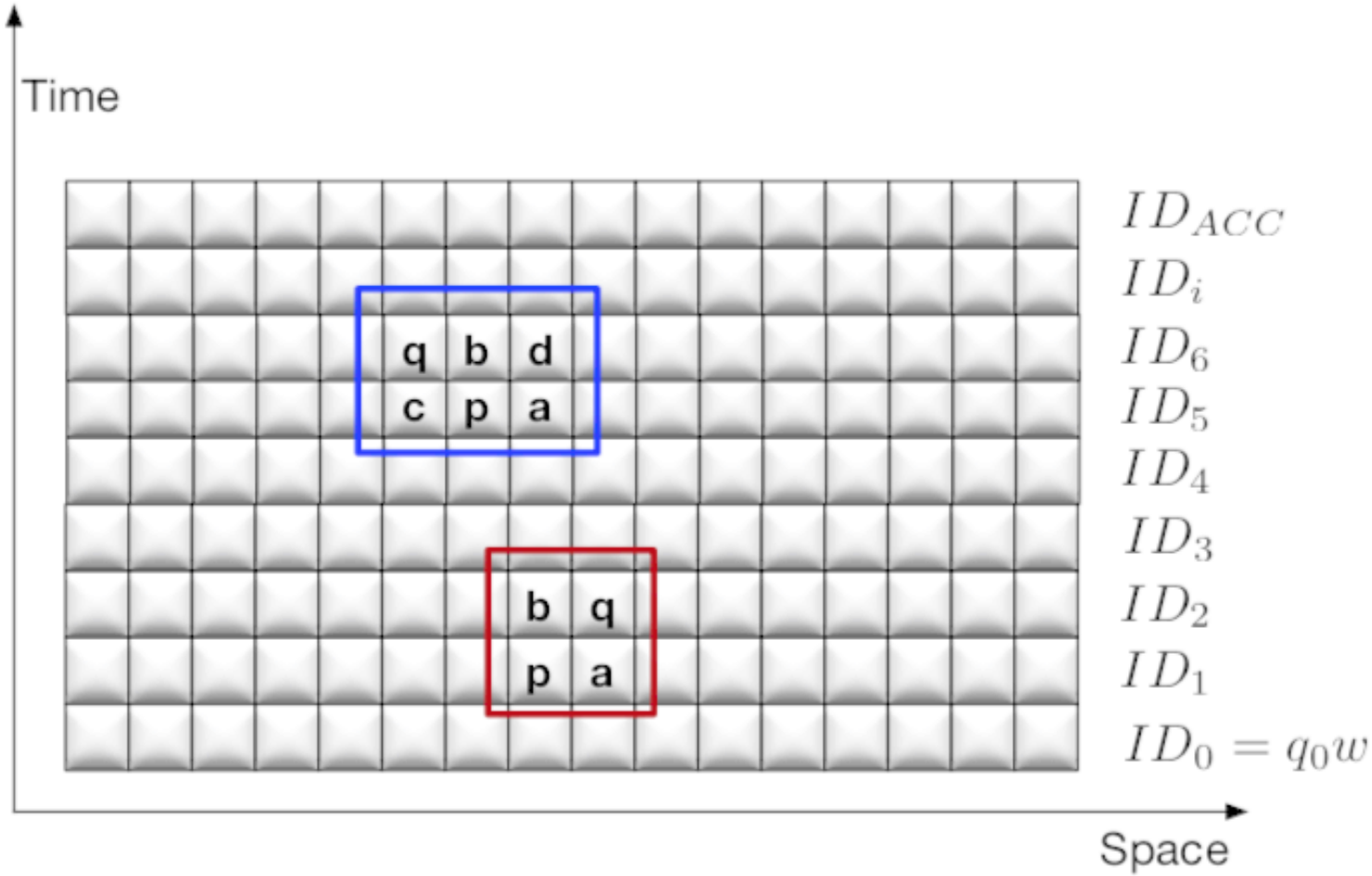
- I.e. A language L is NPC if
  - L is in NP --- has a NP-time algo (guess check on NDTP is P-time)
  - For every problem L' in NP, we have L' <=p L
    - That is, L is harder than anything there is in NP
  - Any problem such as L is "NP-hard" i.e. harder than anything in NP
- So, NPC = NP-hard + in NP
- Finding such an NPC language was the open question that Cook and Levin solved

First NPC problem

- 3SAT was shown NPC as follows
  - First show that 3SAT has an NP algorithm (easy; build an NDTM)
  - Then show that EVERY NP problem has a P-time M.R. to 3-SAT
  - This was achieved by imagining the solution of any NP problem as a collection of “tape histories”
  - Then encoding these histories using 3CNF formulae!

How 3SAT was shown NP-hard (from book)

This is the history of how a TM chugs along. Fortunately, each move from ID_k (full tape) to ID_k+1 (full tape) can be modeled as changes that occur within a 2x2 or 2x3 window. These changes can be captured using a 3CNF formula. …. The rest is history! (see the book for more)



Mapping reductions are key to “connect-up”

- NPC
  - A language L is NPC
    - If L is in NP
      - It has a P-time NDTM
    - EVERY language in NP has a P-time mapping-reduction to L

- In order to show that a NEW language L is NPC
  - We will end up producing a mapping reduction from one of the problems in NPC to L

  - **Then we have a mapping reduction from EVERY language in NP to L**

- **Study this “funnel diagram” (Ch-16) to be convinced**

The “funnel diagram”

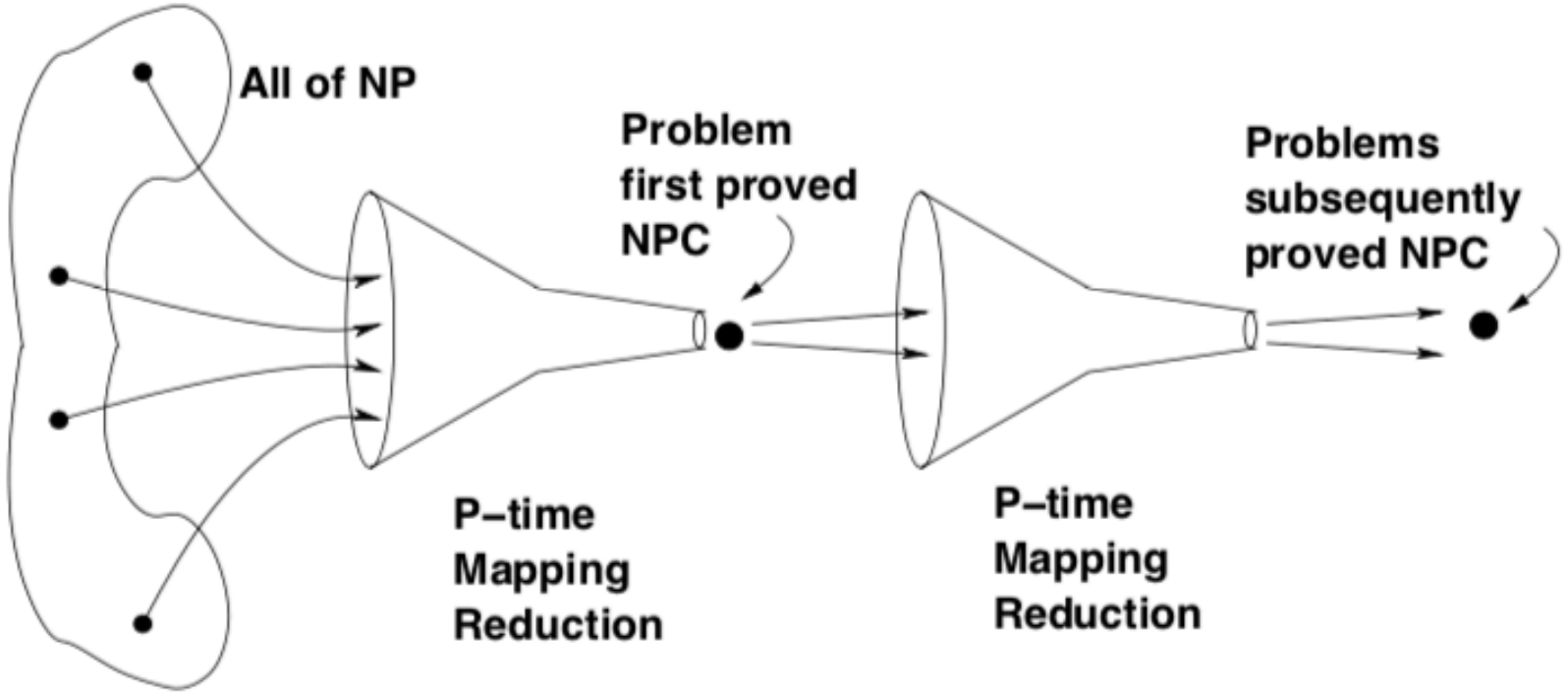


Figure 16.7: Diagram illustrating how NPC proofs are accomplished. The problem first proved NPC is 3-SAT. Definition 16.4(a) is illustrated by the “left funnel” while Definition 16.4(b) is illustrated by the “right funnel.” (The funnels serve as a gentle reminder that mapping reductions need not be onto.)

The “funnel diagram”

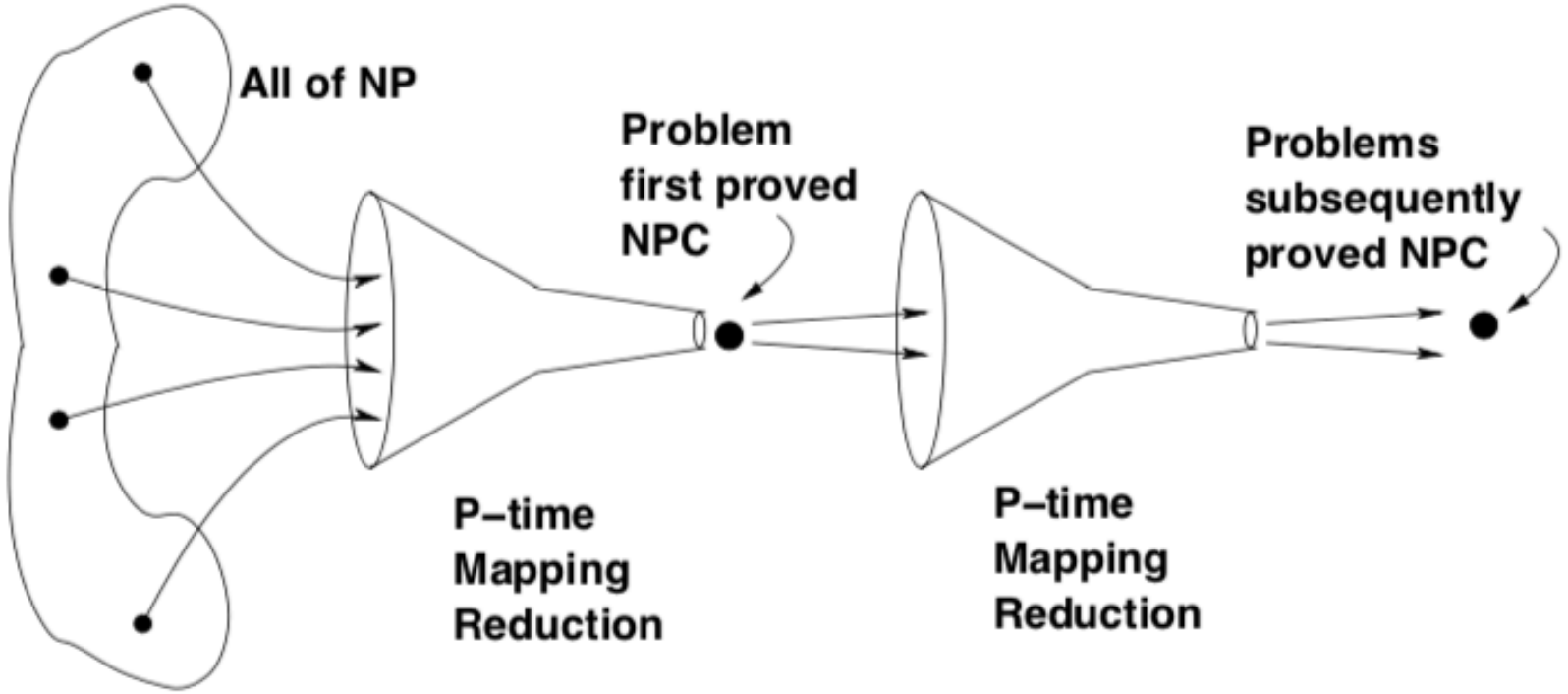
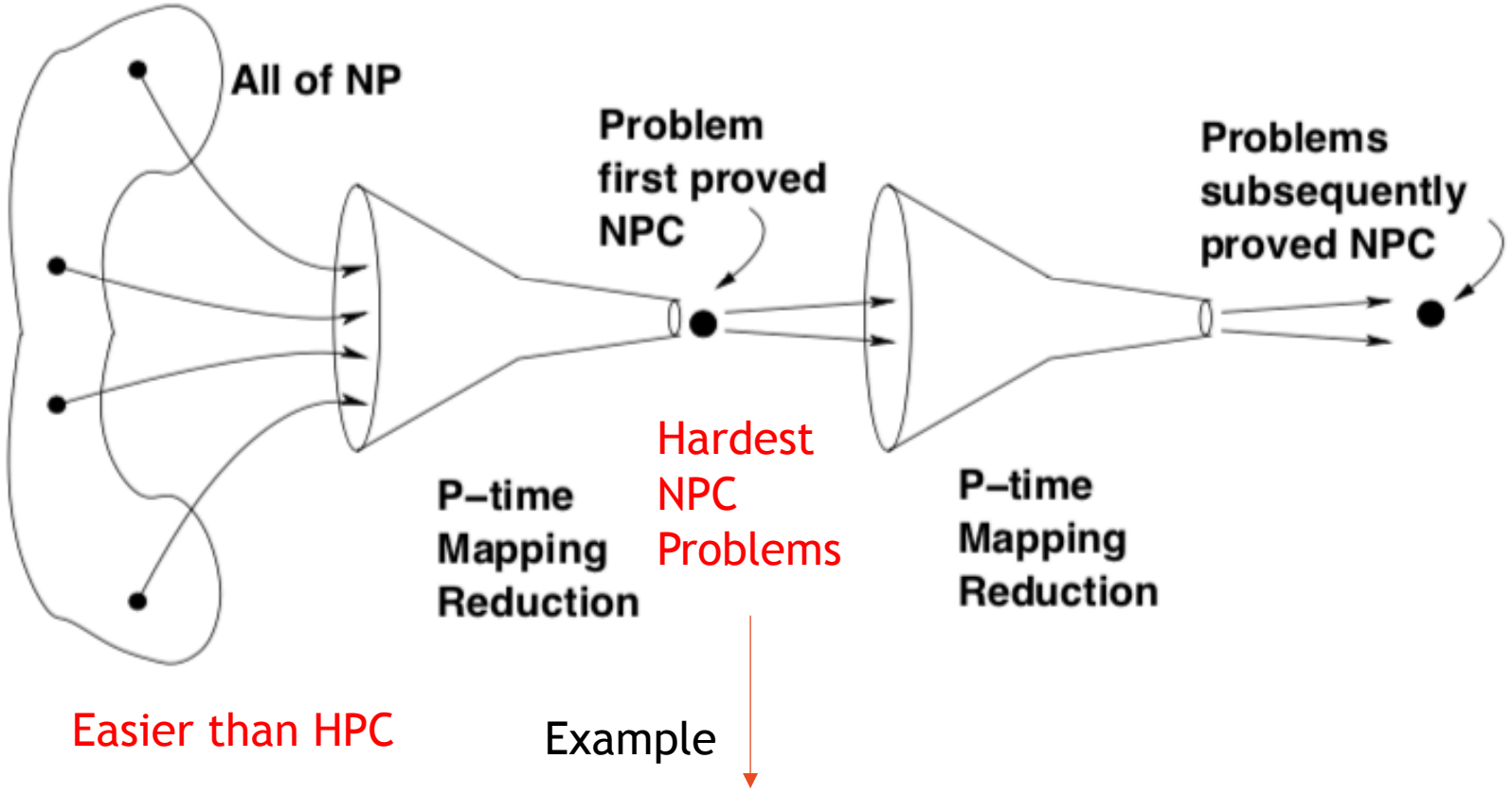


Figure 16.7: Diagram illustrating how NPC proofs are accomplished. The problem first proved NPC is 3-SAT. Definition 16.4(a) is illustrated by the “left funnel” while Definition 16.4(b) is illustrated by the “right funnel.” (The funnels serve as a gentle reminder that mapping reductions need not be onto.)

Easier than HPC

The “funnel diagram”



(a + b + !b) . (!a + a + d) . (a + e + !f)

Figure 16.7: Diagram illustrating how NPC proofs are accomplished. The problem first proved NPC is 3-SAT. Definition 16.4(a) is illustrated by the “left funnel” while Definition 16.4(b) is illustrated by the “right funnel.” (The funnels serve as a gentle reminder that mapping reductions need not be onto.)

The “funnel diagram”

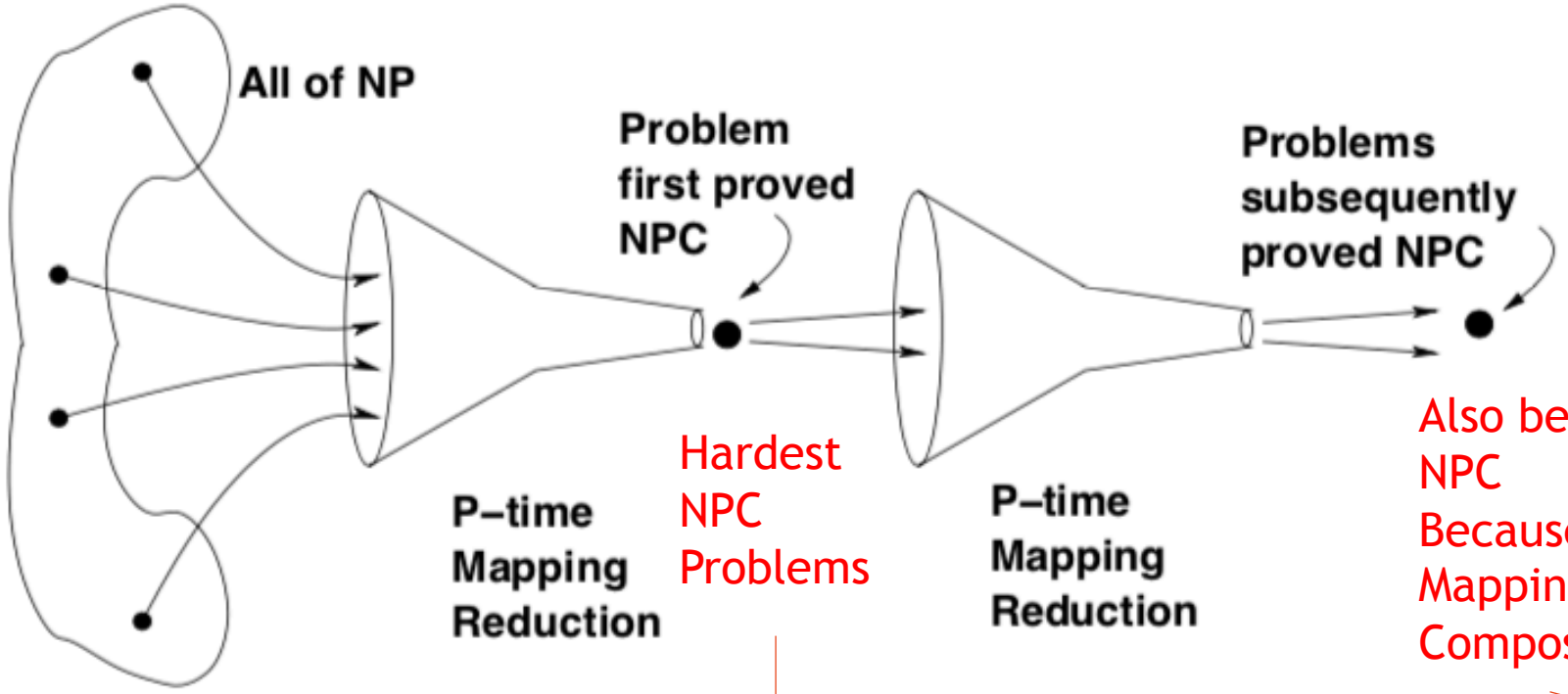


Figure 16.7: Diagram illustrating how NPC proofs are accomplished. The problem first proved NPC is 3-SAT. Definition 16.4(a) is illustrated by the “left funnel” while Definition 16.4(b) is illustrated by the “right funnel.” (The funnels serve as a gentle reminder that mapping reductions need not be onto.)

Hardest NPC Problems

Also becomes NPC Because Mapping-reductions Compose!

Easier than HPC

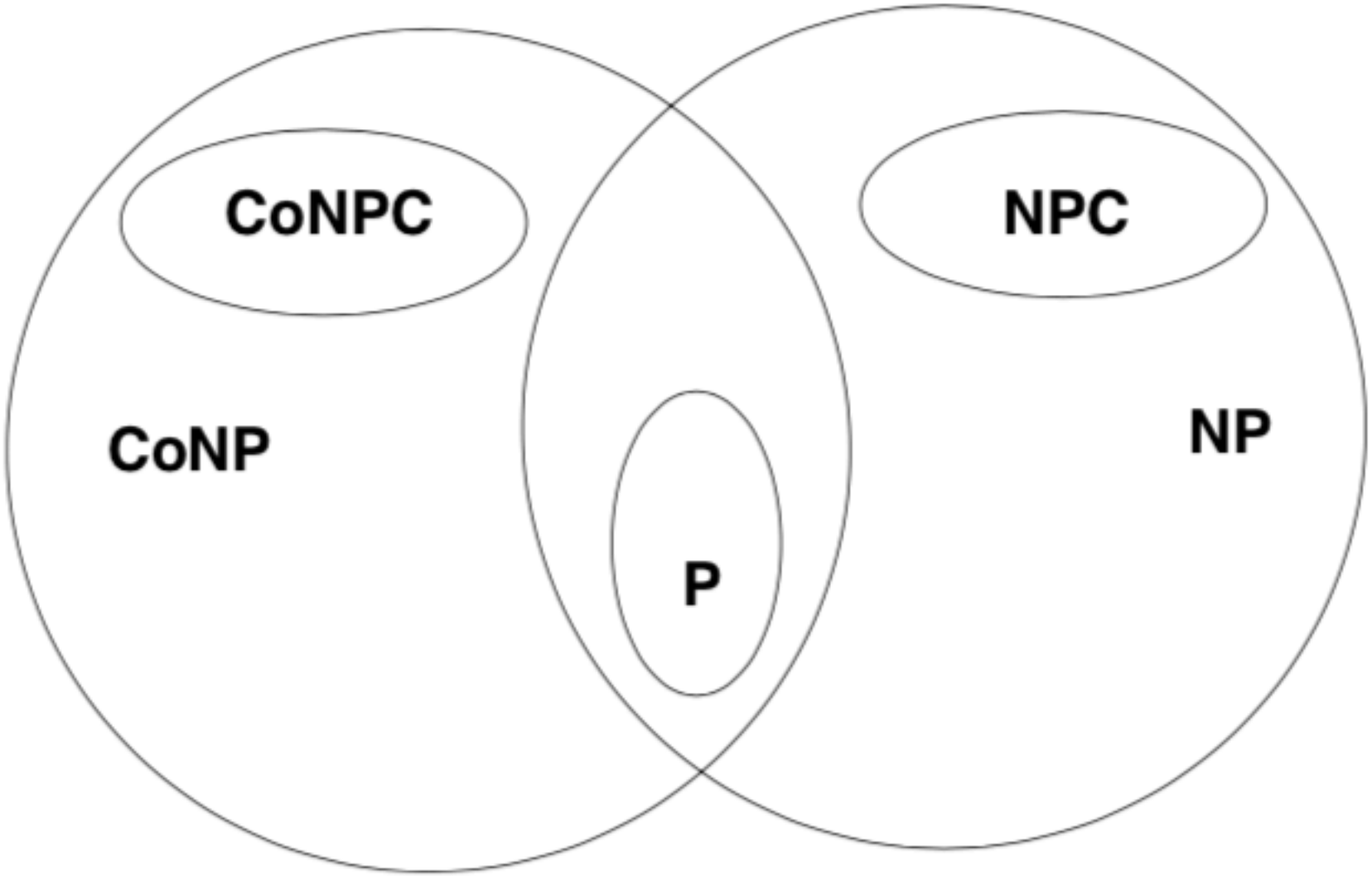
Example

(a + b + !b) . (!a + a + d) . (a + e + !f)

Example

Clique

Language hierarchy in NP-land (ignore “Co” for now)



Boolean Satisfiability: First NPC problem

- From any NDTM, we can compile a 3-SAT formula
- If the NDTM is NP-time, then we can decide the truth of the generated 3-SAT formula in NP-Time

- **If** 3-SAT is deterministic P-time, then we can decide any NDTM in P-time (not in NP-time)

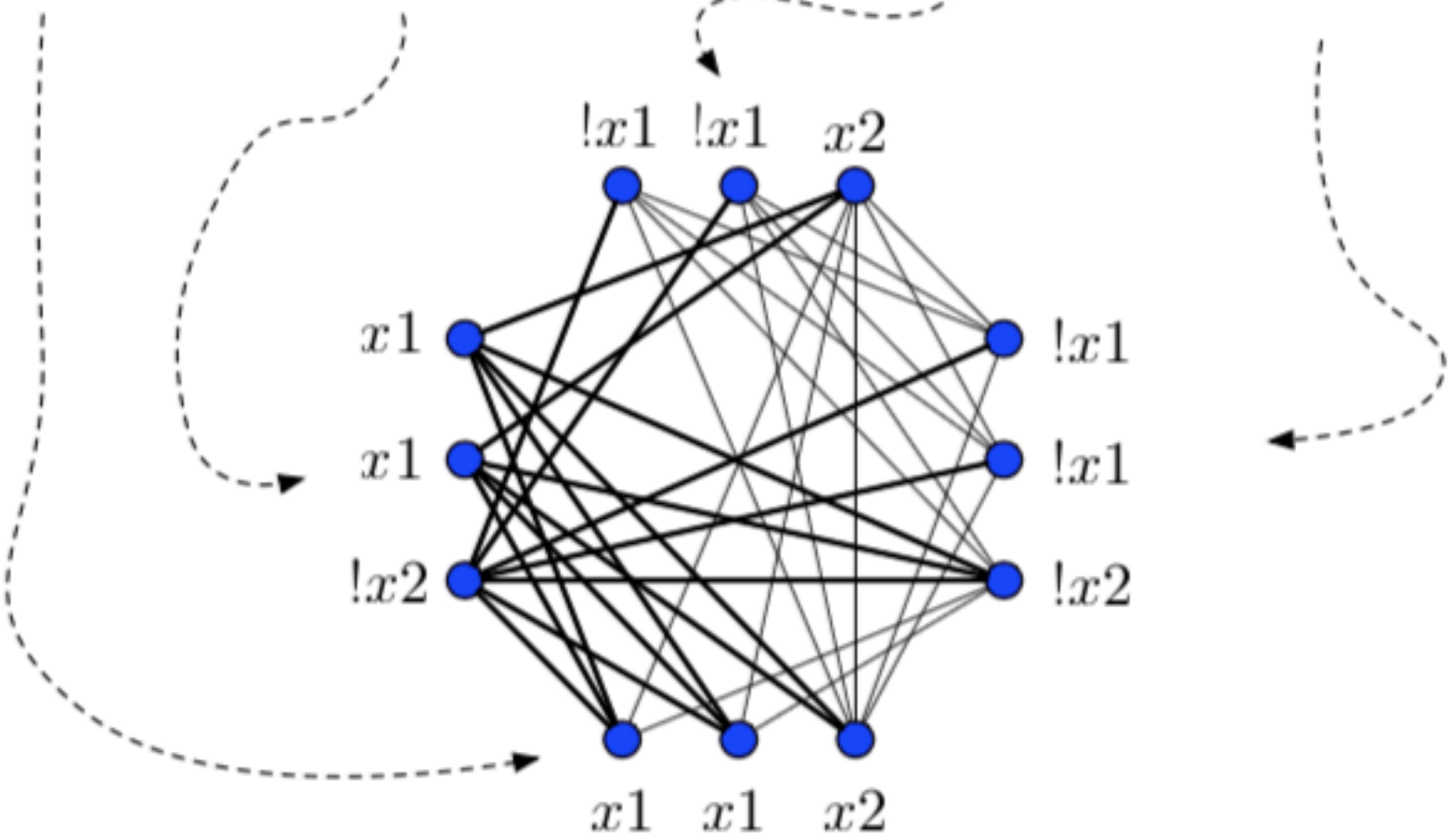
- Details given in the book

Mapping reduction based proof of K-clique being NPC

- Show K-clique is in NPC
  - **Given a description of which nodes are in the clique, we can CHECK that these nodes indeed form a K-clique**
- **Show a mapping reduction from 3-SAT of K clauses into a K-clique instance**

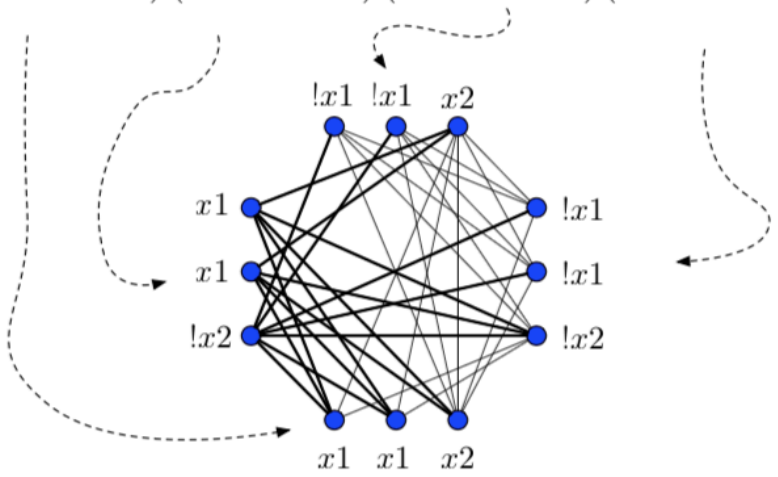
3-SAT to K-clique Mapping Reduction

$$\phi = (x1 + x1 + x2).(x1 + x1 + !x2).(!x1 + !x1 + x2).(!x1 + !x1 + !x2)$$



3-SAT to K-clique Mapping Reduction

$\phi = (x1 + x1 + x2).(x1 + x1 + !x2).(!x1 + !x1 + x2).(!x1 + !x1 + !x2)$



- Depict each clause as an “island” of 3 nodes each
- For a K-clause formula, there are K islands
- Between any two islands, draw an edge from one node of an island to another node of another island

Do the above in ALL possible ways

The edge must connect any two COMPATIBLE nodes

Nodes x and y are compatible if (x and y) is satisfiable

The given K-clause 3-CNF formula is SAT iff there is a K-clique in the mapped graph

This is a classical mapping reduction!

Thus, if there is a P-time algorithm for K-clique, then there is a P-time algorithm for 3-SAT (and ergo for all of NP)

Aside: DNF does not capture the complexity of NPC properly!

$$=3 (x_1 + x_2 + x_3) \cdot (x_1 + !x_2 + !x_3)$$

Given this or any other CNF with N variables, an NDTM can be built such that
 * its first N moves are to write out a variable assignment on the tape
 * then check that under that assignment, the formula is true

But hey, DNF is linear-time SAT-checkable. Multiply the above out to get a DNF

x1 + x1.!x2 + x1.!x3 + x2.x1 + x2.!x2 + x2.!x3 + x3.x1 + x3.!x2 + x3.!x3
→ then simplify

SAT if ANY product-term is ….. ?(what)

This is a linear check
But expansion to DNF turns the formula EXP LONG !!! So no real advantage.

In a sense, DNF is spelling out every possible solution and we have to check one by one !!!