CS 3100, Models of Computation, Spring 20, L27

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

URL: bit.ly/3100s20Syllabus

Agenda

- Go over Asg-7
- Review for the finals
  - Zoom quizzes to help review NPC
- Read MT3AndFinalBatteriesSoln2020.pdf on Canvas
- Basics of Lambda Calculus
  - From First_Jove_Tutorial/CH18/CH18.ipynb

- PLEASE remember to fill your course surveys!
  - Tell us how we could improve, what you learned, etc.

NP-Completeness proofs

Differences between P and NP (Multiple Choice)

- Answer 1: P is the time taken by a DTM to solve a problem
- Answer 2: NP is the time taken by an NDTM to solve a problem
- Answer 3: NP and P may one day be shown the same
- Answer 4: NP definitely means exponential
- Answer 5: P definitely means super-easy to run fast anywhere
- Answer 6: NP is the longest accepting path length in an NDTM
- Answer 7: NP is the shortest accepting path length in an NDTM
- Answer 8: If a problem can be solved in P, it may still be impossible to be solved in NP

Define NP-Hard and NPC (Multiple Choice)

- Answer 1: NP-Hard means too hard for humans
- Answer 2: NP-Hard means too hard for NP standing for a "Number of People"
- Answer 3: NPC means NP-Complete and implies NP-Hard
- Answer 4: X is NP-Hard means all of NP can be P-time mapping reduced to X
- Answer 5: NPC means (1) in NP and (2) NP-Hard
- Answer 6: NP-Hard implies NPC
- Answer 7: NP-Hard problems are always decidable (recursive)
- Answer 8: NPC problems are always decidable (recursive)

How are new problems shown to be NPC these days? "Reduced" means P-time mapping-reduced or <=P (Multiple Choice)

- Answer 1: First step: always show problem is in P
- Answer 2: First step: always show problem is in NP
- Answer 3: Then show that new problem X can be reduced to every NP problem
- Answer 4: Then show that every NP problem can be reduced to X
- Answer 5: Then show that a given NPC problem can be reduced to X
- Answer 6: Then show that X can be reduced to a given NPC problem

X <=p Y means (Multiple Choice)

- Answer 1: Y is at least as hard as X; could be harder
- Answer 2: X is at least as hard as Y; could be harder
- Answer 3: If Y can be solved in P-time then so can X
- Answer 4: If X can be solved in P-time then so can Y

How was the first NPC problem demonstrated to be so (Multiple Choice)

Answer 1: They showed it was in NP (as always)
Answer 2: They reduced an existing NPC problem to the first NPC problem
Answer 3: They showed that any NDTM computation can be encoded as a 3SAT formula
Answer 4: They encoded an arbitrary NDTM computation as a 3CNF formula but also took into account that this NP algorithm can have any P-time execution time

Predicting new non-NPC problems: suppose X is in NP and X is in Co-NP; then how likely is it that X is NPC? How did this help in the past? (Multiple Choice)

- Answer 1: It helped suspect (later find) that Primality checking was in NP
- Answer 2: It helped suspect (later find) that Primality checking was in P
- Answer 3: It helped suspect (later solve) Prime Factorization to be in P
- Answer 4: It helped suspect (later show) that Prime Factorization is in NPC

CNF sat checking is NPC; DNF sat checking is linear (hence P-time). Thus ... (Multiple Choice)

- Answer 1: One can convert CNF to DNF and get a free CNF-sat checking in P time
- Answer 2: One can convert CNF to DNF but formula size can blow-up exponentially; hence no free CNF-sat checking in P time
- Answer 3: DNF does not encode the hardness of problems such as the Traveling Salesperson Problem
- Answer 4: All NPC problems are 3CNF in disguise
- Answer 5: 2CNF is as hard as 3CNF
- Answer 6: 3CNF is as hard as 4CNF
- Answer 7: 3CNF is as hard as X-CNF for any X >= 3

BDD and SAT (Multiple Choice)

- Answer 1: BDDs are minimal DFA (with optimized quick-jumps to acceptance)
- Answer 2: BDDs are minimal DFAs for the satisfying inputs of Boolean functions
- Answer 3: BDD sizes are primarily governed by the chosen variable order
- Answer 4: BDDs are always polynomially sized
- Answer 5: Finding the optimal BDD variable order is always easy
- Answer 6: BDDs must occasionally get exponential or else we would have solved P=NP (as known today)
- Answer 7: SAT tools are quite popular despite SAT being NPC
- Answer 8: One can encode Sudoku, Tetris, Graph Coloring, and Program Static Analysis as SAT
- Answer 9: BDD and SAT are widely used in the industry, especially the semiconductor industry that makes digital systems

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.

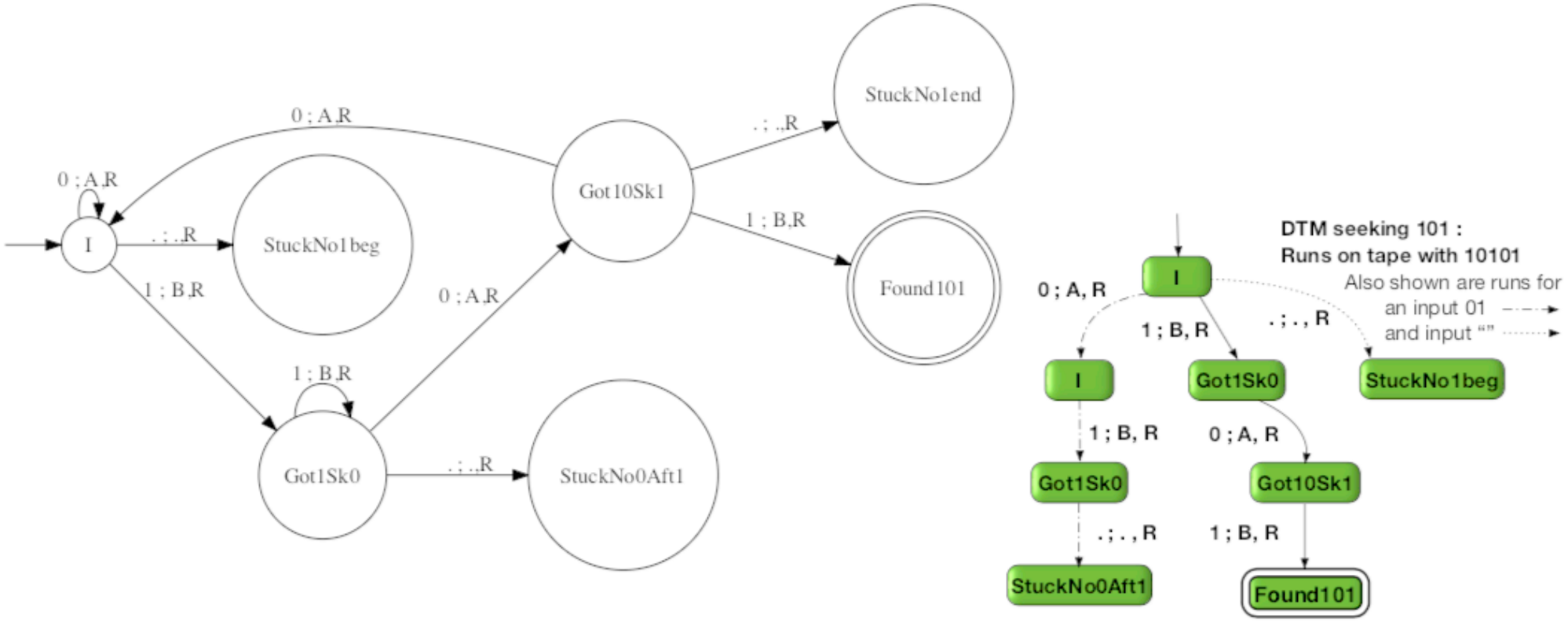
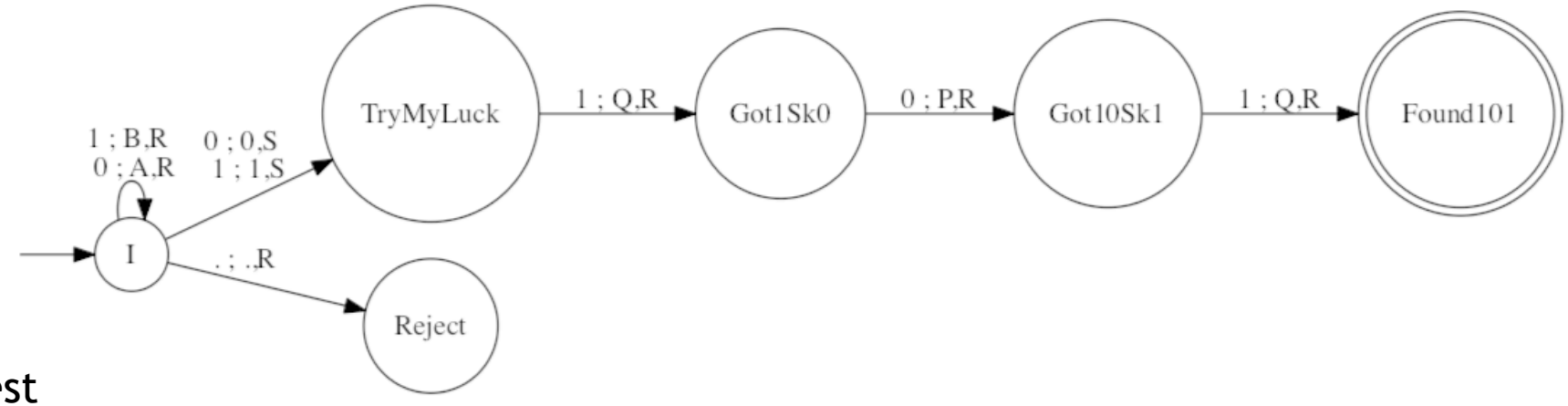


Figure 16.2: Transition diagram and computation tree for a DTM that looks for 101 within given w.

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…)

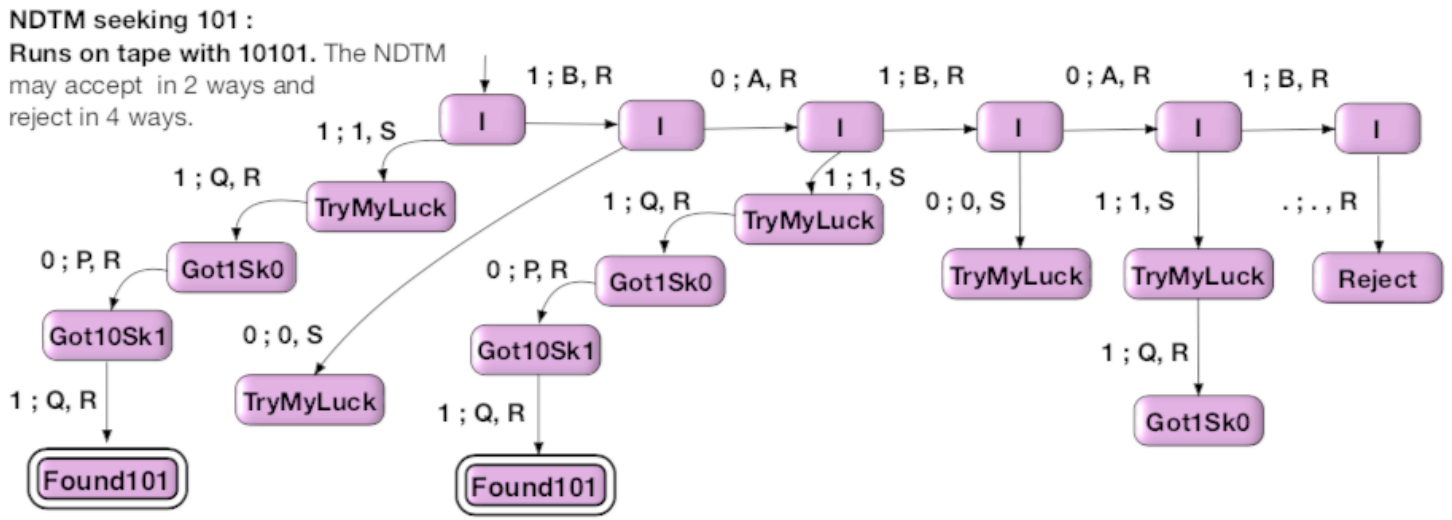
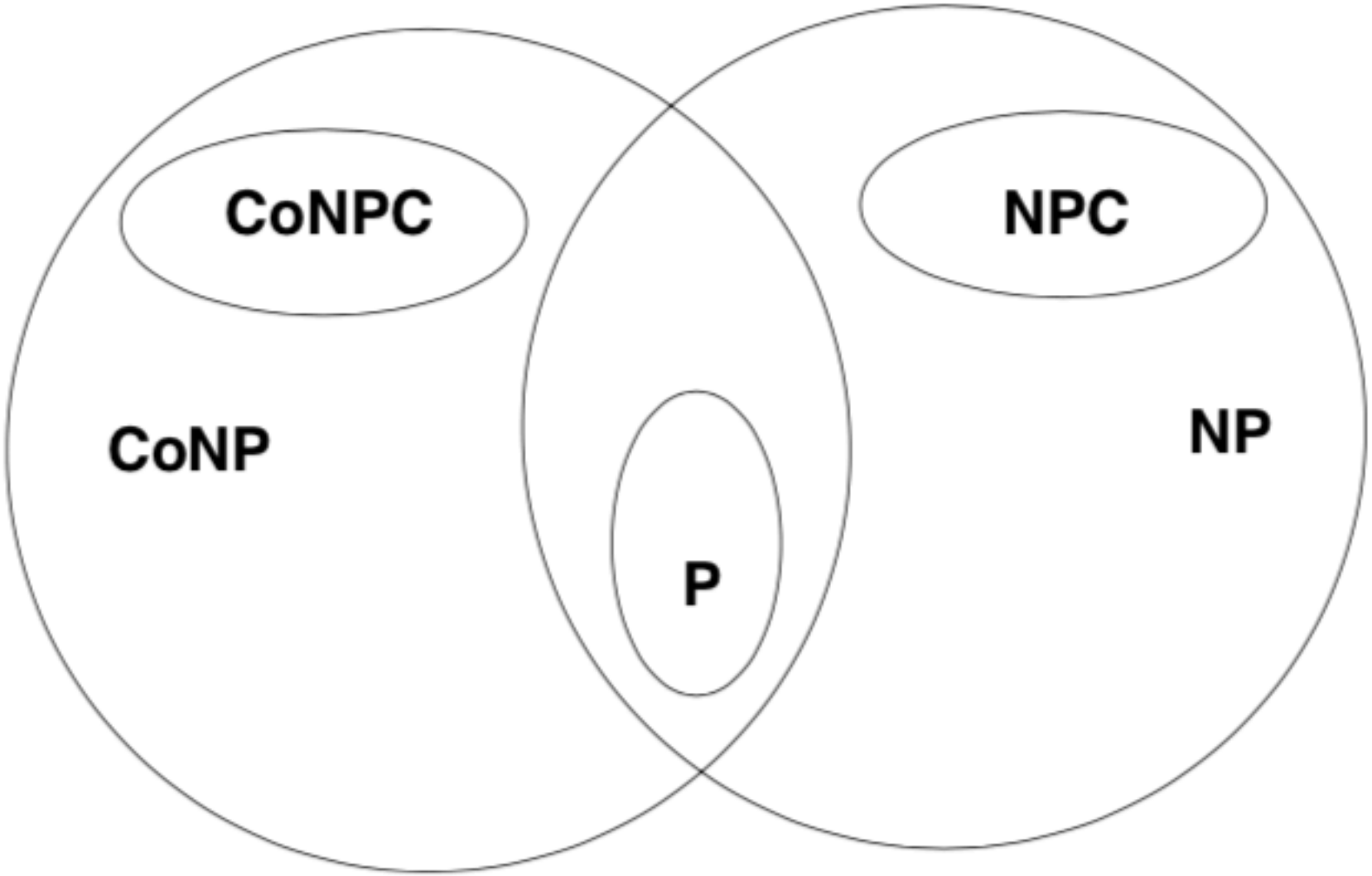


Figure 16.3: Transition diagram and computation tree for an NDTM that looks for 101 within given w.

Definition of 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!

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



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
    - This is hard to do in practice, so we take the practical approach below.

- In order to show that a NEW language L is NPC **in practice**
  - 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”

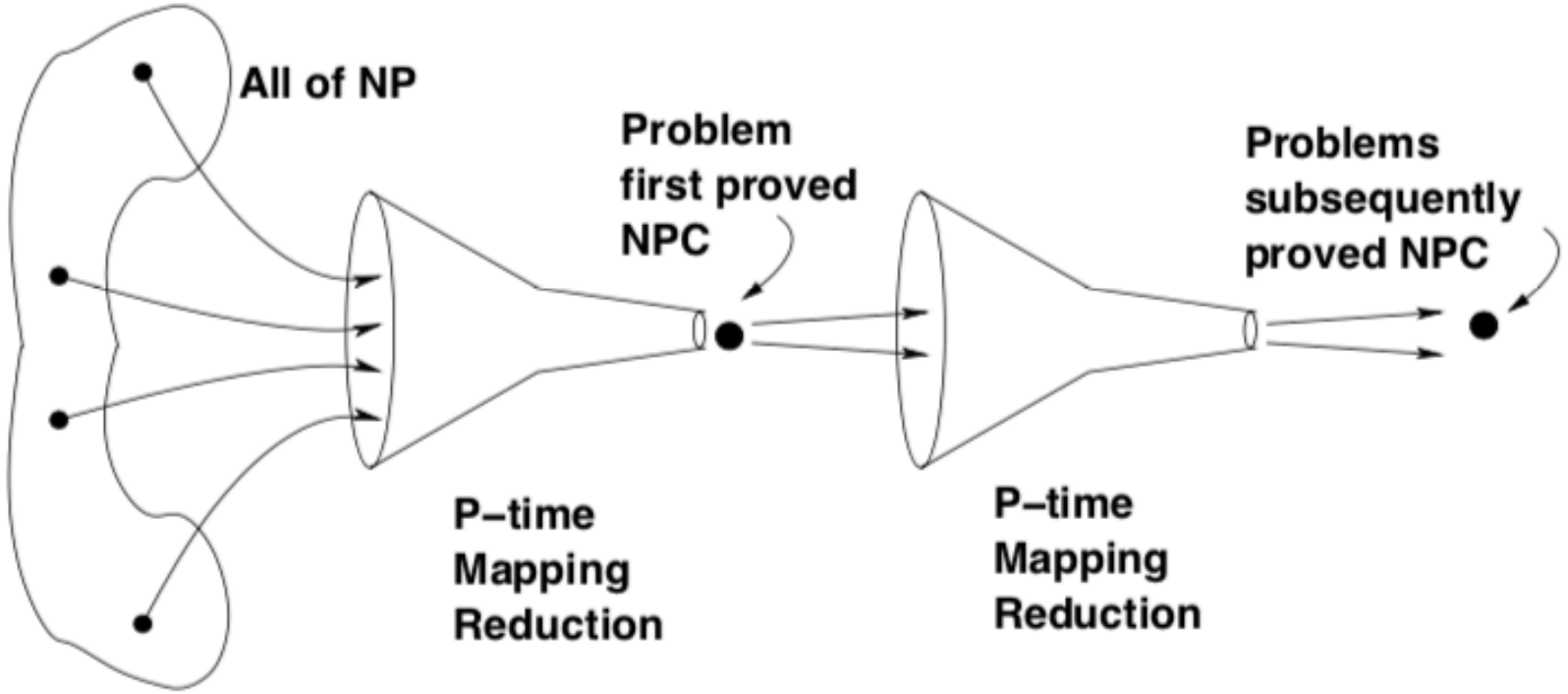


Figure 16.7: Diagram illustrating how NPC proofs are accomplished. The problem first proved NPC is 3-SAT. Definition 16.4(a) is illustrated by the “left funnel” while Definition 16.4(b) is illustrated by the “right funnel.” (The funnels serve as a gentle reminder that mapping reductions need not be onto.)

The “funnel diagram”

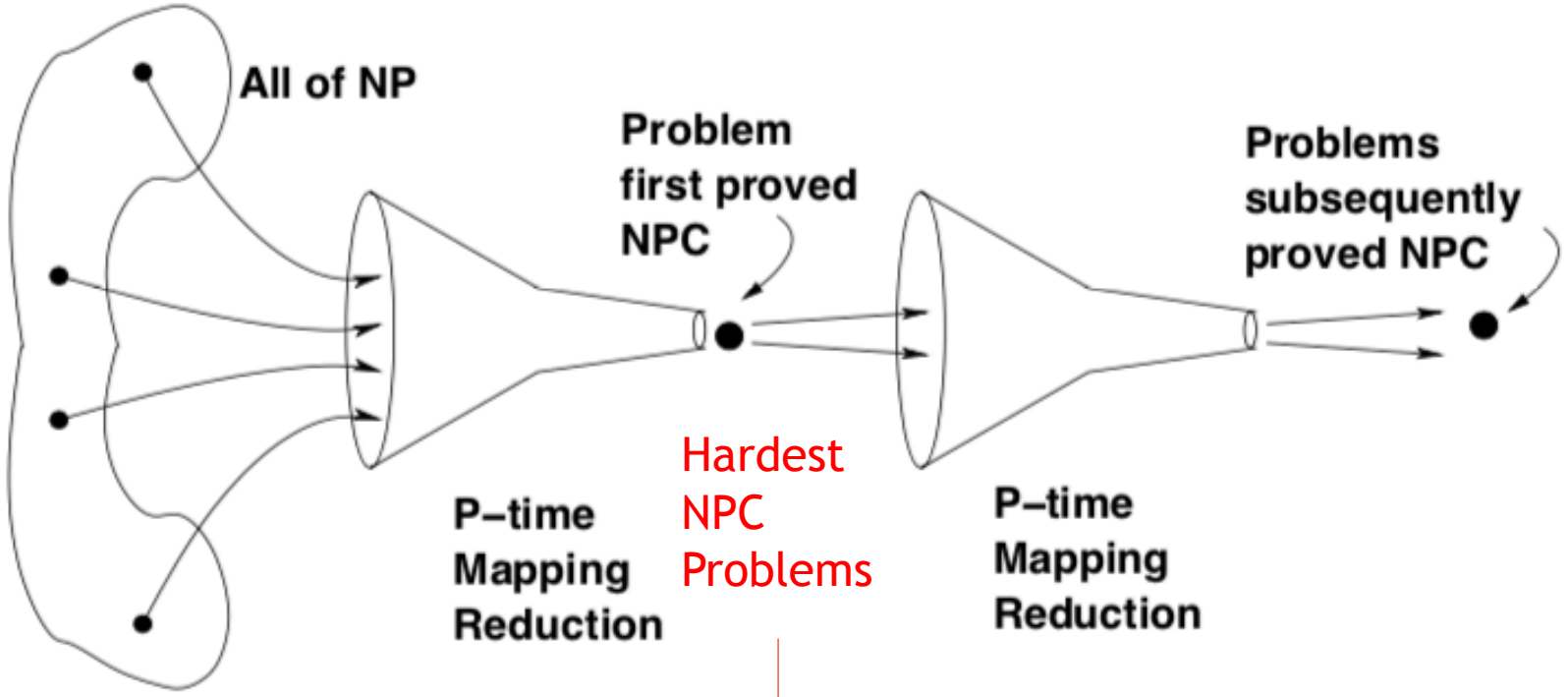


Figure 16.7: Diagram illustrating how NPC proofs are accomplished. The problem first proved NPC is 3-SAT. Definition 16.4(a) is illustrated by the “left funnel” while Definition 16.4(b) is illustrated by the “right funnel.” (The funnels serve as a gentle reminder that mapping reductions need not be onto.)

Example

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

The “funnel diagram”

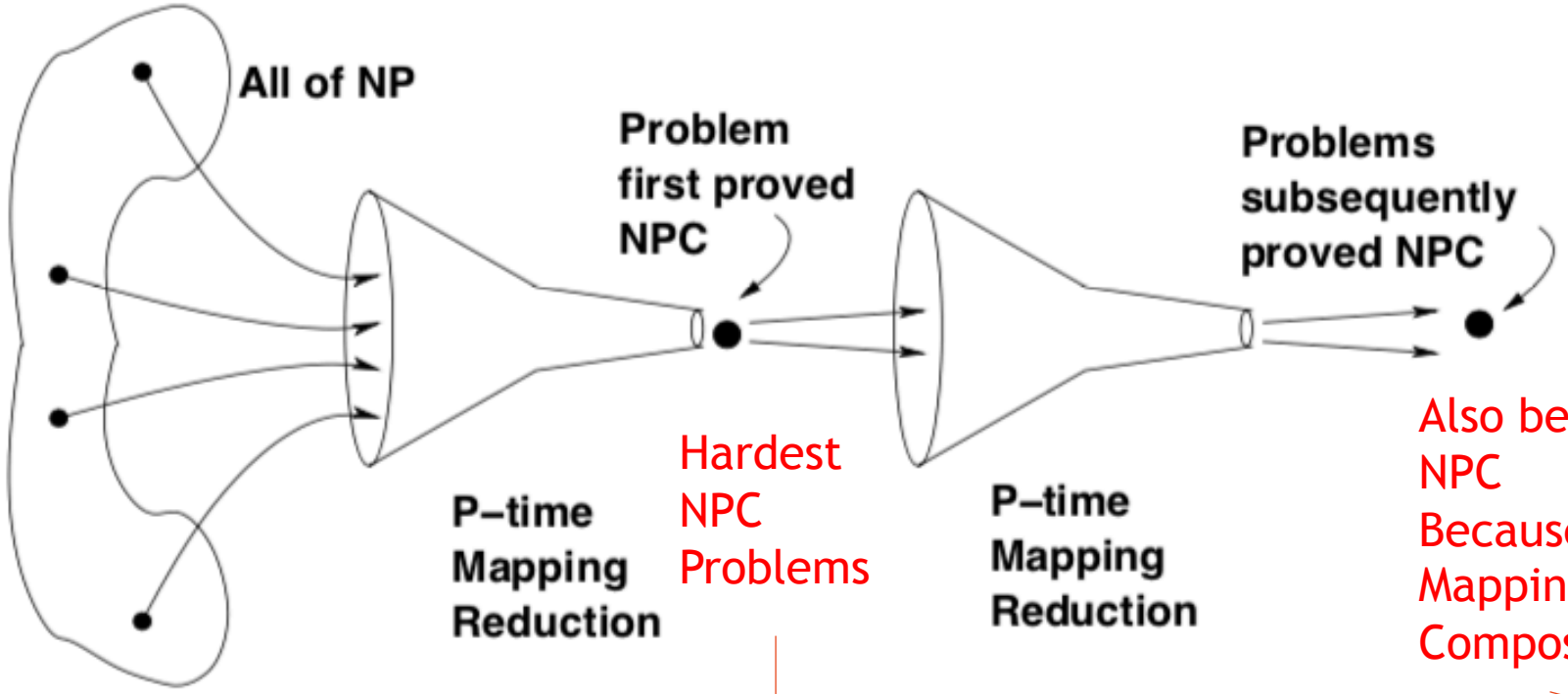


Figure 16.7: Diagram illustrating how NPC proofs are accomplished. The problem first proved NPC is 3-SAT. Definition 16.4(a) is illustrated by the “left funnel” while Definition 16.4(b) is illustrated by the “right funnel.” (The funnels serve as a gentle reminder that mapping reductions need not be onto.)

Hardest
NPC
Problems

Example

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

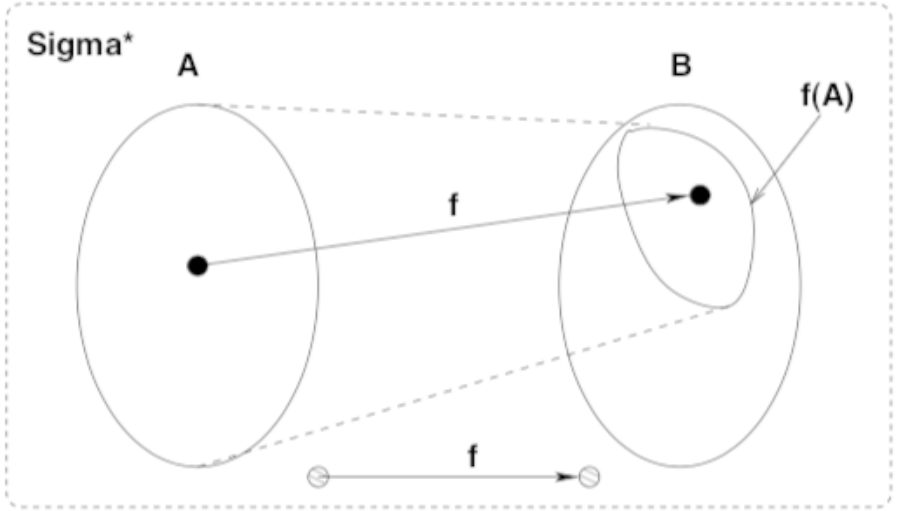
Also becomes
NPC
Because
Mapping-reductions
Compose!

Example

Clique

The basic idea of Mapping Reduction NPC proofs

- A way to take advantage of an already proven hard-fought theorem ("A" below) in establishing the truth of new conjectures ("B" below)
  - We do a proof via diagonalization that **3-SAT is NP-Complete** (the "A" mentioned)
  - We can then handle 1000s of other (often more relevant) questions easily (the "B"s)
    - Prove that "A" <= "B" i
    - i.e. A mapping-reduces-to B **via a Polynomial-time computable function f (i.e. you can code-up f in any of your favorite prog. languages in a way that f NEVER loops on any A-input and runs in P-time. Easy to do - you map Boolean formulae to Graphs, etc as we will see!)**
    - i.e. Given a **P-time solver for B**, we can take ANY A-instance, map it via "f" into a B-instance and feed it to the claimed **P-time B-solver (P-time decider)**



- Given that an MR means

"x in A iff f(x) in B", i.e.

  - x in A => f(x) in B
  - x not in A => f(x) not in B

We can conclude that

"solving B in P-time means Solving A in P-time --> A Turing Awa

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.)

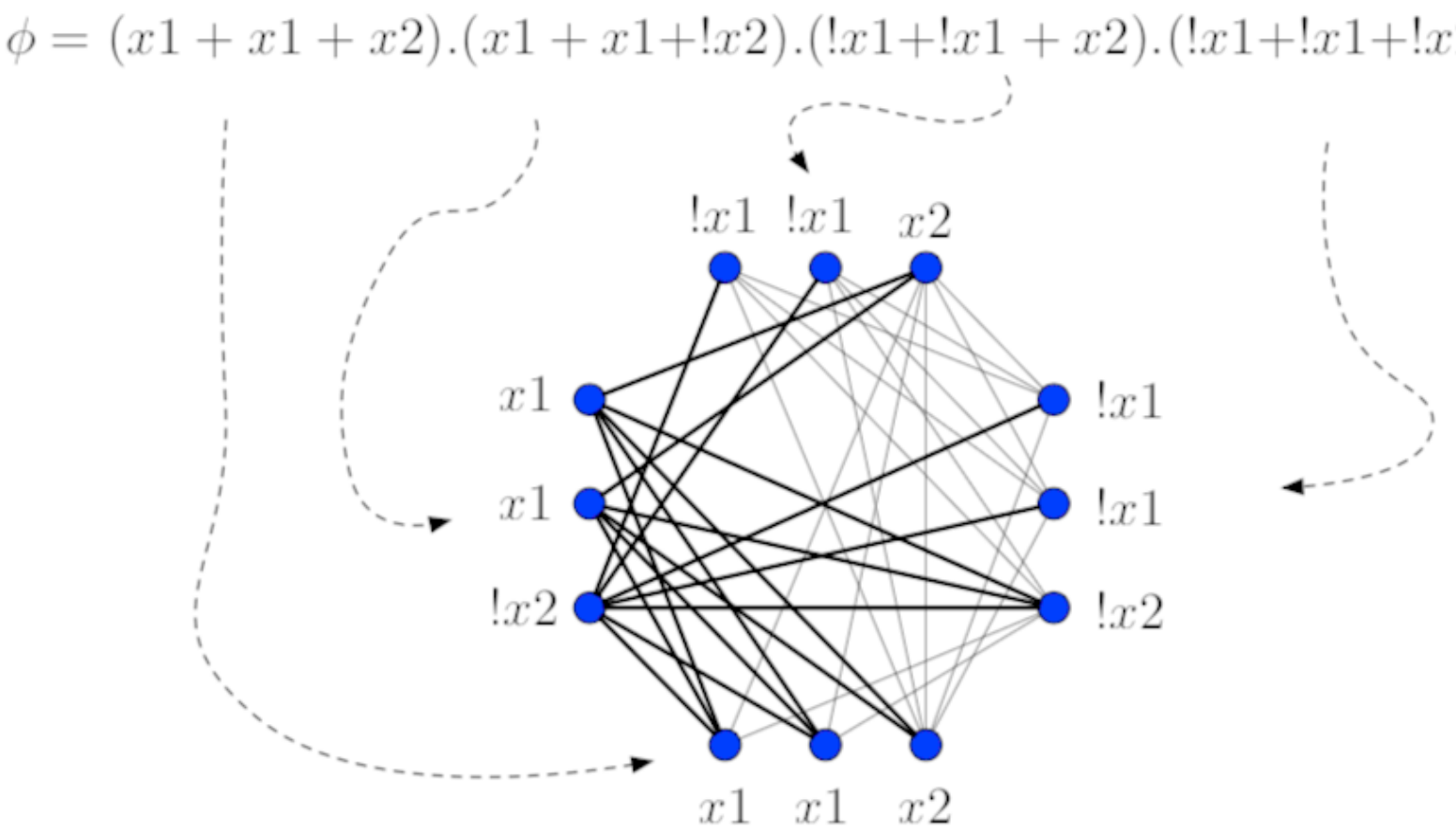


Figure 16.9: The Proof that *Clique* is NPH using an example formula $\varphi = (x1 + x1 + x2).(x1 + x1 + !x2).(!x1 + !x1 + x2).(!x1 + !x1 + !x2)$. We never connect the nodes within each clause "island" (there are four such islands, each with three nodes). Across each clause island, we draw edges in 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.

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 "
- You'll also gain exposure to Boolean Satisfiability (SAT tools)
- BDD and SAT are industrial tools (used for verifying chips and SW to be bug-free wrt deep properties you can state)
  - I gave you a SAT demo on 4/13/20. You can solve Asg-7 qns using that.
- SAT is fun too (for solving Sudoku etc)
- https://medium.com/@rvprasad/sat-encoding-solving-simpler-sudoku-d926f1206d1e

Binary Decision Diagrams

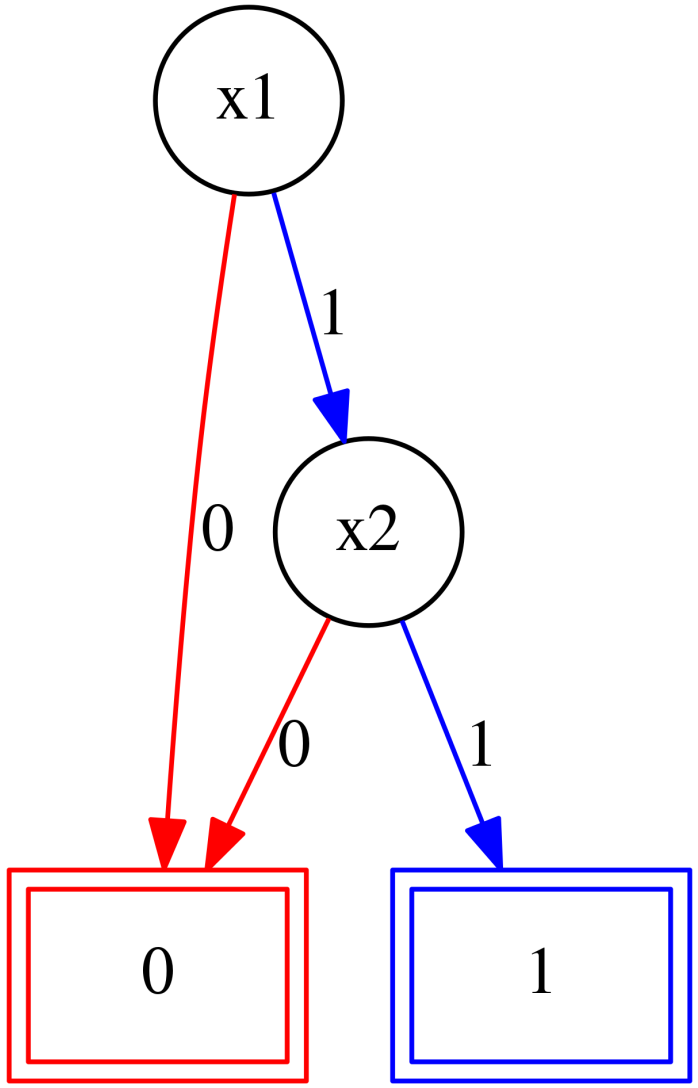
BDDs

- 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

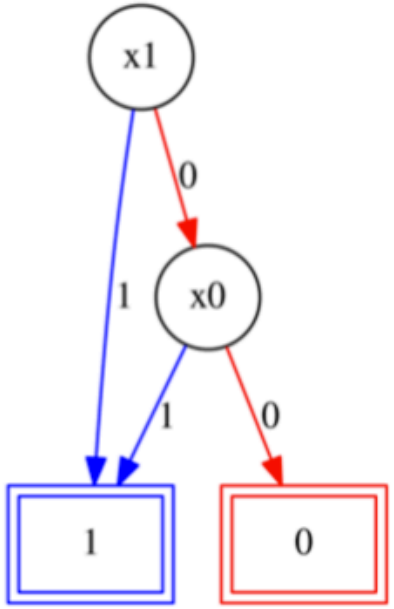
```
# http://formal.cs.utah.edu:8080/pbl/BDD.php
# Very simple example to show-off syntax

#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
```



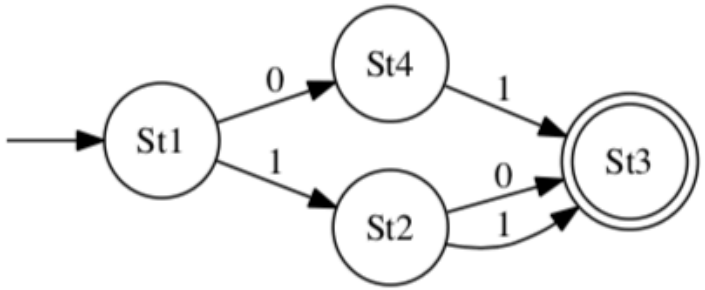
Things to observe about BDDs

- 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

BDDs are minimal DFA in disguise



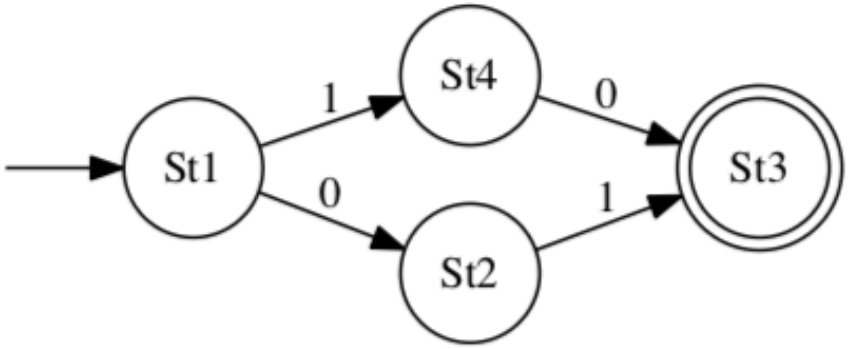

```
L_OR  = "(01+10+11)" # Regexp for the on-set of the OR function
dotObj_dfa(min_dfa(nfa2dfa(re2nfa(L_OR))), STATENAME_MAXSIZE=4)
```



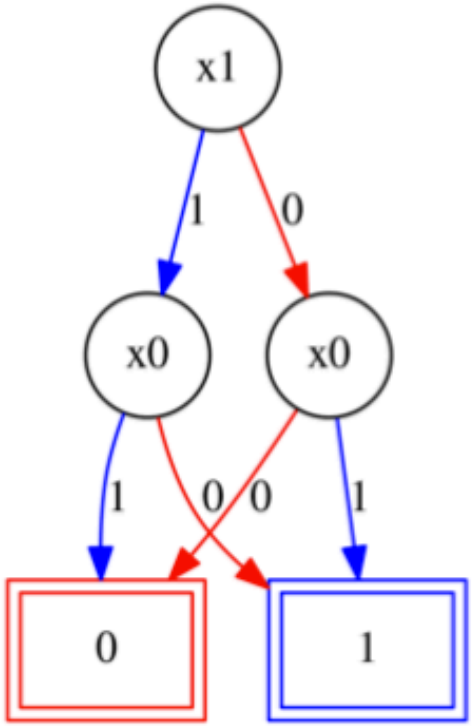
By entering these commands and clicking "build BDD," one obtains the Or BDD of Figure 17.5.

BDDs are minimal DFA in disguise

```
L_XOR = "(01+10)" # The regexp for the on-set of the XOR function
dotObj_dfa(min_dfa(nfa2dfa(re2nfa(L_XOR))), STATENAME_MAXSIZE=4)
```



We can see that the minimal DFA[3] for Xor accepts 01 and 10. The BDD for Xor can be obtained using an online tool called PBDD[4] that can be invoked as follows (it will open the BDD tool in a new browser tab):



Good vs Bad BDD var order (poly vs exp size)

study how to build a BDD for x2x1x0 < y2y1y0

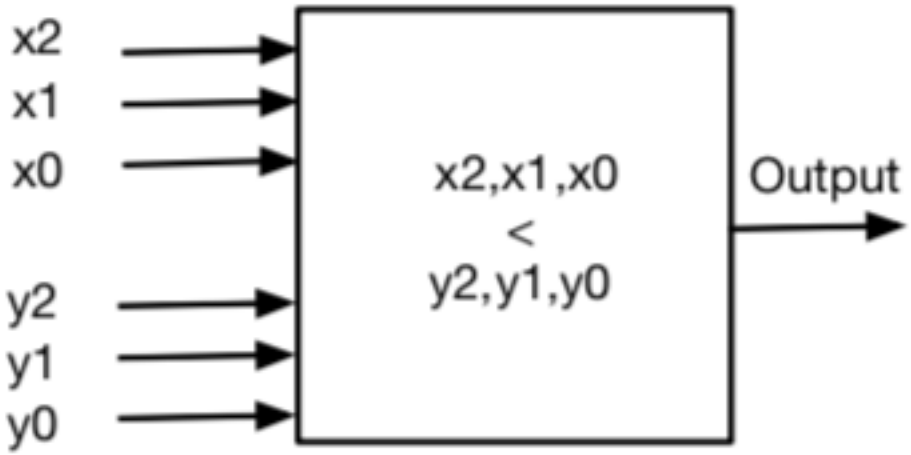


Figure 17.6: The < comparator.

Good vs Bad BDD var order (poly vs exp size)

study how to build a BDD for x2x1x0 < y2y1y0 : All satisfying Boolean combinations as a regexp → give it to Jove !!

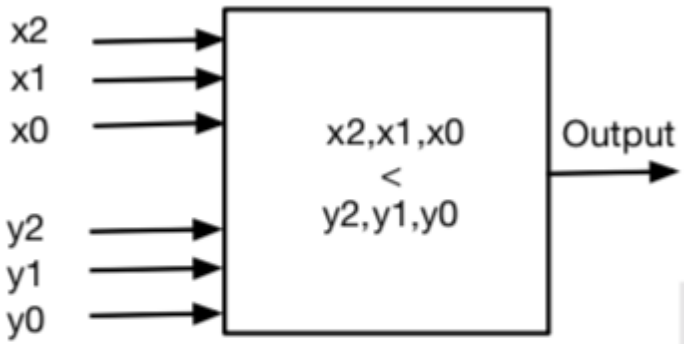


Figure 17.6: The < comparator.

```
R =  "(000001+000011+000111+001011+001111+010011+010111+011111+\
100101+100111+101111+110111+000010+000101+000110+001010+001101+\
001110+010101+010110+011101+011110+100110+101110+000100+001100+\
010100+011100)"
```

R express a “bad order” (not bad odor) → we fed in order of x2 x1 x0 THEN y2 y1 y0

A quick decision is facilitated by x2 y2 x1 y1 x0 y0 (coming later)

Bad BDD var --> blowup

study how to build a BDD for x2x1x0 < y2y1y0 : All satisfying Boolean combinations as a regexp → give it to Jove !!

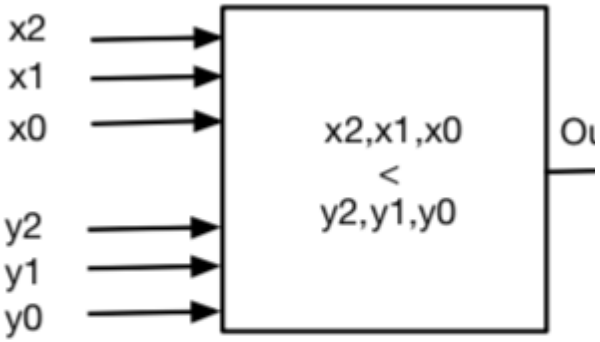


Figure 17.6: The < comparator.

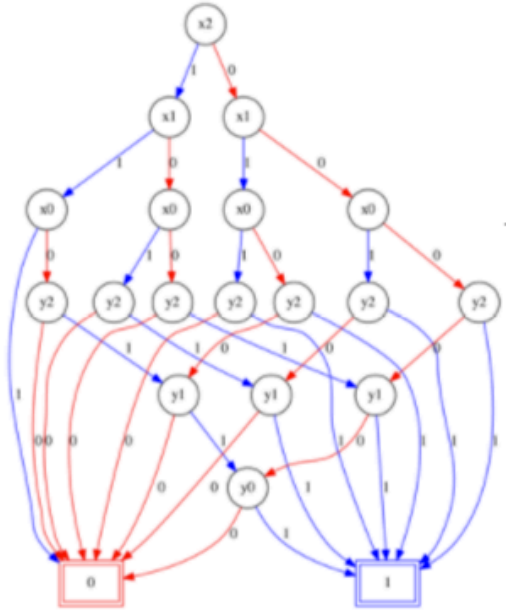
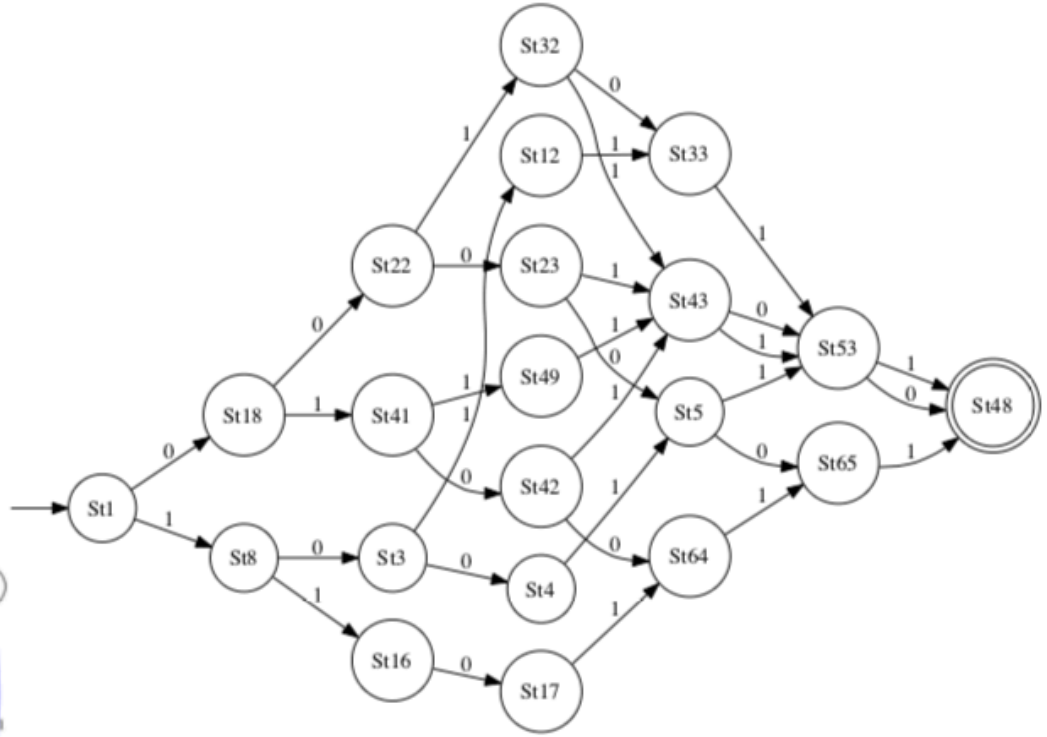


Figure 17.7: BDD for the magnitude comparator: bad input-variable order



The minimal DFA for R using the method shown earlier is above, and the BDD for it is in Figure 17.7 (for `Var_Order` being `x2 x1 x0 y2 y1 y0`). This DFA is in fact exponential in the x2,x1,x0 bits (those are the first three bits to arrive at this machine, and the machine grows exponentially with respect to these inputs). It must represent every x2,x1,x0

Good BDD var --> no blowup

study how to build a BDD for x2x1x0 < y2y1y0 : All satisfying Boolean combinations as a regexp → give it to Jove !!

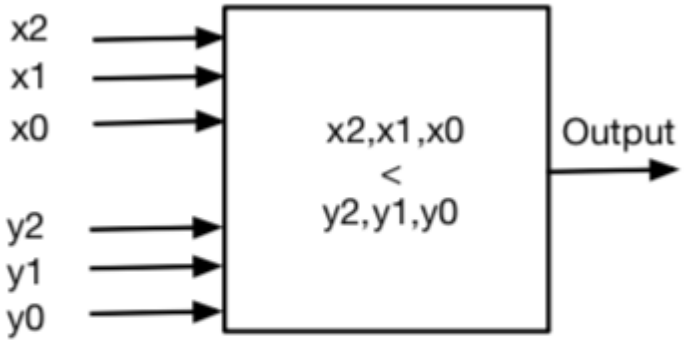
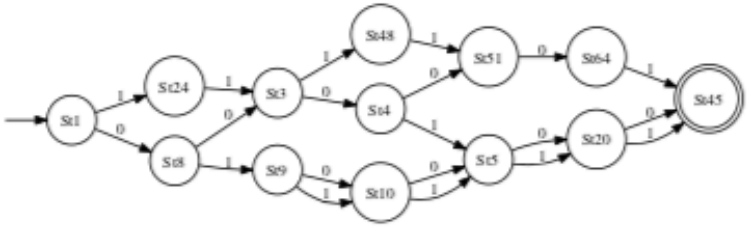


Figure 17.6: The < comparator.

Let us call the regular expression obtained by interleaving the input bits "Rmix":

```
Rmix="(000001+000111+001101+011111+110001+110111+111101+000101+\
000110+010111+011011+011101+011110+110101+110110+000100+010011+\
010101+010110+011001+011010+011100+110100+010001+010010+010100+\
011000+010000)"
```

The minimal DFA for Rmix is below, and the BDD for it is in Figure 17.8 (for `Var_Order` being `x2 y2 x1 y1 x0 y0`). Again, it is easy to see that the BDD does not "trudge through" the redundant decodings. For instance, in the DFA, state St8 is reached when $x_2 = 0$, and then when $y_2 = 1$ is seen, a pathway of redundant decodings leading to the accept state is entered. Correspondingly, in the BDD, after seeing $x_2 = 0$, we reach a node which decodes y_2, and if $y_2 = 1$, the BDD jumps to the "1" leaf node.



17.3.2 Functions with linearly sized BDDs

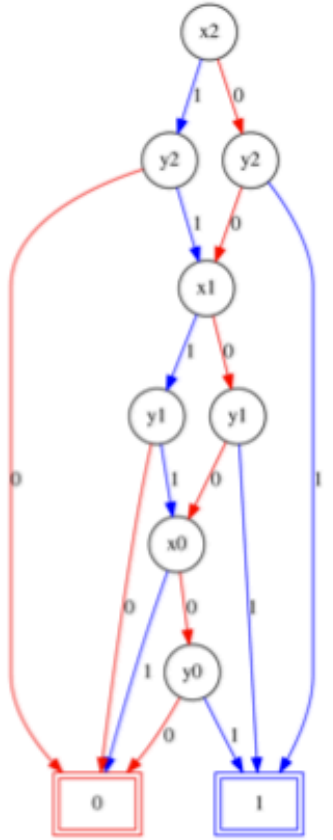


Figure 17.8: BDD for the magnitude comparator: good input-variable order

Good BDD var --> no blowup for multi-input xor

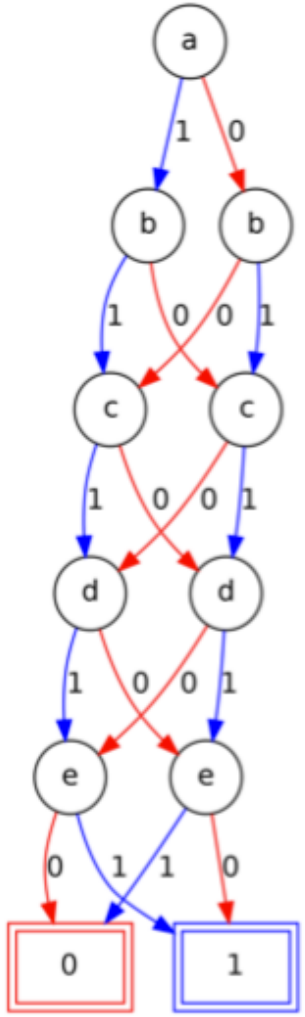


Figure 17.9: BDD for a 5-input *Xor* gate

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?

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

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

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

This is the online interface for a BDD package written using PBL.

About this BDD implementation | **Language Spec**

ENTER YOUR FORMULA HERE

```
Var_Order : x1 x2 x3
Main_Exp :  (x1 | x2 | x3) & (!x1 | !x2 | !x3) & (!x1 | !x2 | !x3) & (!x1 | x2 | !x3)
```

PROGRAM OUTPUT

```
Number of satisfying assignments: 5
Number of Variables : 3
Number of Nodes     : 6
Variable Ordering
-----------------------------------
['x1', 'x2', 'x3']

All satisfying assignts:
------------------------------
[0, 0, 1]
[0, 1, 1]
[1, 1, 0]
[0, 1, 0]
[1, 0, 0]
```

Prev | Next | Build BDD | Build minimum BDD

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

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

This is the online interface for a BDD package written using PBL.

About this BDD implementation | **Language Spec**

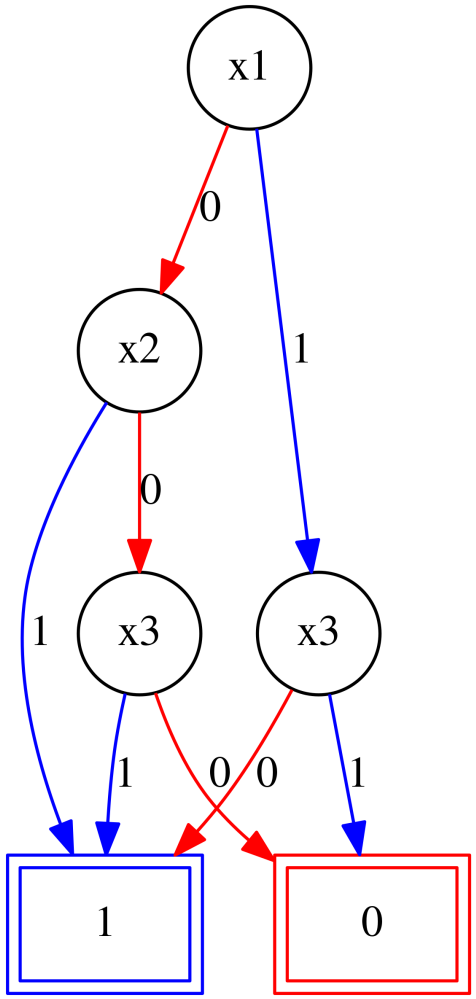
ENTER YOUR FORMULA HERE

```
Var_Order : x1 x2 x3
Main_Exp :  (x1 | x2 | x3) & (!x1 | !x2 | !x3) & (!x1 | !x2 | !x3) & (!x1 | x2 | !x3)
```

PROGRAM OUTPUT

```
Number of satisfying assignments: 5
Number of Variables : 3
Number of Nodes     : 6
Variable Ordering
-----------------------------------
['x1', 'x2', 'x3']

All satisfying assignts:
----------------------------
[0, 0, 1]
[0, 1, 1]
[1, 1, 0]
[0, 1, 0]
[1, 0, 0]
```

Prev | Next | Build BDD | Build minimum BDD



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: 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

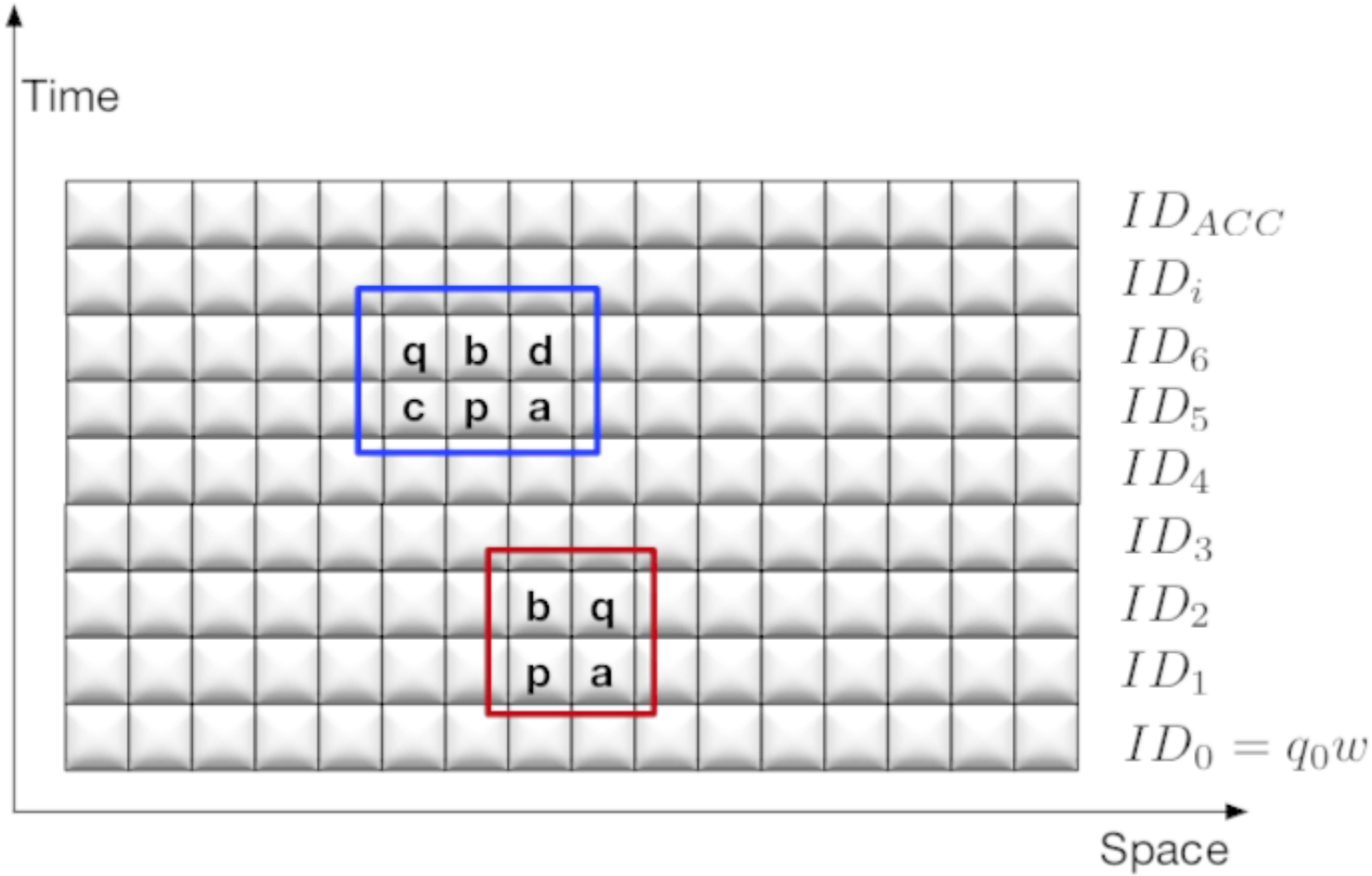
- I.e. A language L is NPC if
  - L is in NP --- has a NP-time algo (guess check on NDTP is P-time)
  - For every problem L' in NP, we have L' <=p L
    - That is, L is harder than anything there is in NP
  - Any problem such as L is "NP-hard" i.e. harder than anything in NP
- So, NPC = NP-hard + in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



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

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 !!!

How does it relate to primality testing?

- General result
  - Researchers believe that it is highly unlikely that a language is NPC and the complement of that language is in NP
  - I.e. NPC and Co-NP IFF NP = Co-NP (highly likely)

  - So if a problem is in NP and the complement of the problem is in NP also
    - i.e. L in NP and N in Co-NP
  - Then it is unlikely that L is NPC

- This allows people to predict that some algos are “easy”
- This happened in 1977 and 2002
  - Primes are in NP (1977)
  - Primes are in P (2002) because Primes were in NP and Co-Primes were also in NP; so unlikely that Primes were NPC. This was established in 2002
- Note this is only primality testing – not Prime Factorization – which is still of unknown hardness!