CS 480 Syllabus and Portfolio Winter 2019

Bryan Muller

April 3, 2019

Portfolio

Course Tracker

You are required to track your progress in the course using this table.

Note: Currently, you see full credit for week one's work. (✓ means yes. Blank means no.) Update the table for week 2 and all subsequent weeks each class day and week during the semester.

Week	CRU	PFP	CDL	SAQ	PAQ	CDL	PPL
1	✓	✓	✓	✓	✓	✓	100%
2	✓	✓	✓	✓	✓	✓	100%
3	✓	✓	✓	✓	✓	✓	85%
4	✓	✓	✓	✓	✓	✓	100%
5	✓	✓	✓	✓	✓	✓	100%
6	✓	✓	✓	✓	✓	✓	100%
7	✓	✓	✓	✓	✓	✓	100%
8	✓	✓	✓	✓	✓	✓	100%
9	✓	✓	✓	✓	✓	✓	100%
10	✓	✓	✓	✓	✓	✓	80%
11	✓	✓	✓	✓	✓	✓	100%
12	✓	✓	✓	✓	✓	✓	100%
13	✓	✓	✓	✓	✓	✓	100%

This is an honest and true record of my work for this course.

Signature: _____

Grade Claims

On the week indicated, bring this updated document to my office and make your claim.

Claim Week	Grade Claim	Instructor Grade	Adjusted Grade
5	A	A	A
9	A	A	A
13 - 14	A		

Evidences

Fill in your evidences here each week to build your portfolio. The number of pieces of evidence are determined by you. However, the more you have the better off you will be.

Week 9

Chapter 14 Exercises

1 $TM_{INFINDFA} = \{ \langle D \rangle : D$ is a DFA and L(D) is an infinite language $\}$

If L(D) is infinite, $TM_{INFINDFA}$ accepts If L(D) is not intinite, $TM_{INFINDFA}$ rejects

Using the theorem $B_D = \{ w \in \Sigma^* : w \in L(D)$ and $p \leq |w| \leq 2p \}$ where p is the number of states and Σ is the alphabet of DFA D, then L(D) is infinite if and only if $B_D \neq$.

Using this theorem, we can construct a TM that decides $TM_{INFINDFA}$.

- TM = On input $\langle D \rangle$, where D is a DFA
- Let p be the number of states in DFA $\langle D \rangle$.
- Let Σ be the alphabet of DFA D.
- For each string $w \in \Sigma^*$ such that $p \leq |/w/| \leq 2p$ do: a. Run $TM_{DFA}(\langle D, w \rangle)$ b. If it accepts then accept
- Otherwise Reject.

2 Given LBA with tape of length S, alphabet A, and number of states Q, then the number of possible configurations of the LBA is QSA^S. If the machine does halt, it will halt within QSA^S iterations, otherwise the machine is stuck in a loop by virtue of the pigeonhole principle.

We can create a Turing Machine to run the LBA upto QSA^S steps and determine if it has halted within that many steps. If so, accept, otherwise reject.

3 $Halt_{TM} = \{\langle M, w \rangle$: M is a TM and w its input and M halts on $w\}$

- Keep listing pairs $\langle M, w \rangle$ from Σ^* on an "internal tape"
- Keep checking whether M is a Turing machine description. If so, M happens to be a Turing machine description.
- Run Turing machine M on w, treating w as its input. Do not run to completion; instead engage in a dovetailed execution with all other TMs and inputs meanwhile being enumerated internally.
- When the dovetailed simulation finds an $\langle M, w \rangle$ pair such that M halts on w it lists the $\langle M, w \rangle$ pair on the output tape.

This listing will produce every $\langle$ M, w $\rangle$ such that M halts on w. The existence of this enumerator means that Halt_{TM} is RE.

4 Describe an enumerator for the language '$\text{L}_{\text{UnivCFG}}$
'$\text{L}_{\text{UnivCFG}} = \{\langle$ G $\rangle$: G is a CFG and L(G) $\neq \Sigma^*$ $\}$

- Keep listing $\langle$ G $\rangle$ from Σ^* on an "internal tape"
- Keep checking whether G is a CFG. If so, G happens to be a CFG description
- Begin running G on strings $\langle$ w $\rangle$ from Σ^*, treating w as it's input. Do not run to completion, instead engage in a dovetailed execution with all other CFGs and inputs meanwhile being enumeratied internally.
- When the dovetailed simulation finds an $\langle$ G, w $\rangle$ pair such that w $\neg$ $\in$ L(G) it lists $\langle$ G $\rangle$ on the output tape.

5 Semi-decider for whether or not a grammar G_1 has a language that is *not* contained in the language of another grammar G_2.

- Keep enumerating strings from $\text{L}(\text{G}_1)$
- Feed each enumerated string w into G_2
- If G_2 rejects w, return true ($\text{L}(\text{G}_1) \not\subset \text{L}(\text{G}_2)$)

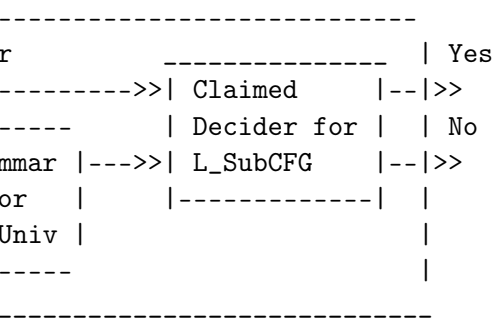
This is a semi decider, because both $\text{L}(\text{G}_1)$ and $\text{L}(\text{G}_2)$ could be infinite in size with $\text{L}(\text{G}_1) \subseteq \text{L}(\text{G}_2)$ being true. If this is the case, the semi-decider will enter into an infinite loop and never return anything, without being able to return false, as it hasn't checked every string in $\text{L}(\text{G}_1)$ against $\text{L}(\text{G}_2)$.

6

- Suppose there is a decider SubCFG(G_1, G_2) that can determine whether grammar G_1 generates a langage $\text{L}(\text{G}_1)$ which is a subset of the langugage $\text{L}(\text{G}_2)$ generated by grammar G_2. Suppose SubCFG(G_1, G_2) will halt and emit "yes" exactly when every string in $\text{L}(\text{G}_1)$ has been found in $\text{L}(\text{G}_2)$.
- Ask the user for a grammar G_{User}
- Create the Turing machine SubCFG(G_{User}, G_{Univ}) where G_{Univ} is a grammar for the universal language UnivCFG.

- If SubCFG returns "yes", then G_{Univ} has a universal language - this is known to be impossible.
- Therefore, SubCFG cannot exist.

Decider for SubCFG



7 L_{AmbCFG} not RE

Recursively Enumerable languges are not closed under complementation. This means that the complement of a language L is only Recursively Enumerable if and only if L is also recursive.

If L and 'L are both Recurseively Enumerable languages, then L is decidable. L is undecidable $\rightarrow$ L is not R.E. || 'L is not R.E.

Week 10

Monday March 11

Mapping Reducability f : $\Sigma^* \rightarrow \Sigma^*$ is a computable function if some Turing Machine M on every input w halts with just f(w) on its tape.

Example Mapping Reduction

1. Let A be a language known to be undecidable ("old" or "existing" language).
2. Let B be the language that must be shown to be undecidable ("new" language).
3. Find a mapping reduction f from A to B.
4. If B has a decider D_B then we can decide membership in A as follows

Reduce Old to New N<O? NO!

Friday March 15

Theorems 0-1

- 0 A language is decidable iff it is Turing-recognizable and co-Turing-recognizable RE = Turing recognizable
- 1 If A $\leq_{\mathrm{m}}$ B and B is Turing-recognizable, then A is Turing-recognizable A is known language, B is unknown lanaguage
- 2 If L is a regular language, and A is mapping reducible to it, then A is decidable.

Chapter 15 Exercises

15.2.3.1

- Solution 1

  Length is 75

  PCP Solver Solution

```
[1, 3, 3, 1, 1, 2, 1, 3, 3, 3, 2, 1, 1, 1, 3,
 2, 1, 3, 3, 2, 1, 3, 2, 1, 1, 3, 2, 1, 3, 1,
 3, 3, 2, 1, 3, 2, 1, 1, 3, 3, 1, 1, 3, 2, 1,
 3, 2, 2, 1, 1, 2, 2, 1, 3, 2, 2, 2, 2, 3, 3,
 2, 2, 3, 3, 1, 2, 1, 1, 2, 3, 2, 2, 3, 2, 2]

100 1   1   100 100 0   100 1   1   1   0   100 100 100 1
1   0   0   1   1   100 1   0   0   0   100 1   1   1   0

0   100 1   1   0   100 1   0   100 100 1   0   100 1   100
100 1   0   0   100 1   0   100 1   1   0   100 1   0   1

1   1   0   100 1   0   100 100 1   1   100 100 1   0   100
0   0   100 1   0   100 1   1   0   0   1   1   0   100 1

1   0   0   100 100 0   0   100 1   0   0   0   0   1   1
0   100 100 1   1   100 100 1   0   100 100 100 100 0   0

0   0   1   1   100 0   100 100 0   1   0   0   1   0   0
100 100 0   0   1   100 1   1   100 0   100 100 0   100 100
```

- Solution 2

  Length is 75

  PCP Solver Solution

```
[1, 3, 3, 1, 1, 2, 1, 3, 3, 3, 2, 1, 1, 1, 3,
 1, 3, 3, 2, 2, 1, 2, 1, 2, 2, 3, 1, 3, 3, 3,
 3, 1, 2, 1, 1, 1, 1, 2, 3, 2, 1, 3, 3, 2, 2,
 3, 1, 3, 2, 1, 2, 1, 1, 3, 3, 1, 3, 2, 3, 2,
 2, 1, 1, 2, 1, 3, 2, 2, 2, 3, 2, 2, 3, 2, 2]
100 1   1   100 100 0   100 1   1   1   0   100 100 100 1
1   0   0   1   1   100 1   0   0   0   100 1   1   1   0

100 1   1   0   0   100 0   100 0   0   1   100 1   1   1
1   0   0   100 100 1   100 1   100 100 0   1   0   0   0

1   100 0   100 100 100 100 0   1   0   100 1   1   0   0
0   1   100 1   1   1   1   100 0   100 1   0   0   100 100

1   100 1   0   100 0   100 100 1   1   100 1   0   1   0
0   1   0   100 1   100 1   1   0   0   1   0   100 0   100

0   100 100 0   100 1   0   0   0   1   0   0   1   0   0
100 1   1   100 1   0   100 100 100 0   100 100 0   100 100
```

15.2.3.4 a. Done in Jove

b.

i.

Given this scenario ($len(T_i[0] < len(T_{i1})$ or $T_i[0] > len(T_i[1])$ for all T_i), then there will never be a solution due to the fact that with this constraint one side of the string will never be able to catch up to the other.

ii. This condition indicates that for at least two of the tiles, have a top where $|T_j[0]| < |T_j[1]|$ and $|T_k[0]| > T_k[1]|$

Given the properties of the GCD and LCM, we can prove that there is an infinite number of solutions to a diophantine with the following constraints: $Mx + Py = Nx + Ry$ and $M > N$ and $P < R$

Therefore, we can create a decider for a unary PCP that behaves like the following:

Given a set of tiles, T: if $\exists$ t $\in$ T : |t[0]| == |t[1]| -> accept (we have found a trivial solution)

if $\forall$ q,r $\in$ T : (|q[0]| < |r[1]|) $\vee$ (|q[0]| > |r[1]|) -> reject (one side will never catch up to the other)

if $\exists$ q,r $\in$ T : (|q[0]| < |r[1]|) $\wedge$ (q[0]| > |r[1]|) -> accept (some combination of tiles will result in a match)

15.5.1.3 Mapping reduction from Halt_{TM} to A_{TM}

- Let HALT_{TM} = { $\langle$ M, w $\rangle$: M halts on input w}
- Assume HALT_{TM} is decidable.
- Assume there is a decider for A_{TM} called D_{ATM}(M', w) where

```
D_ATM(M', /w/):
  accepts => M' accepts /w/ => M accepts /w/

  rejects => M' rejects /w/ => M rejects /w/ or loops
```

- Now construct a decider for HALT_{TM} D_{HALTTM}(M', w) where

```
D_HALTTM(M', /w/):
   accepts => M' accepts /w/ => M accepts or rejects /w/

   rejects => M' rejects /w/ => M loops
```

- We can now redefine an algorithm for D_{ATM}:

```
Run D_HALTTM on \lang M, /w/ \rang:
  If D_HALTTM rejects => reject
  If D_HALTTM accepts => continue
Run M on /w/:
  If M accepts => accept
  If M rejects => reject
```

- This has created a contradiction. If a decider for the Halt_{TM} could be created, then we could create a decider for A_{TM}. We know A_{TM} is undecidable, therefore a decider for Halt_{TM} cannot exist.

15.5.1.5

```
 CFL_TM = { \lang M \rang : M is a TM whose language is context-free. }

 A_TM to CFL_TM

 A \le_m CFL_TM

Decider_CFLTM (M')

   M'(x) (where x is form \lang M, /w/ \rang{
      if x is of the form a^{n}b^{n}c^{n} accept
      Run M on /w/
         If M accepts /w/ => accept
         If M rejects /w/ => reject
   }
```

Week 11

Chapter 16.7.2 Exercises

Exercise 4

```
(setq formula2 '(((a) b) ((b) c) ((c) d) ((d) a)))
(to-dimacs formula2)
```

p cnf 4 4 -4 1 0 -1 2 0 -2 3 0 -3 4 0

This is satisfiable via CryptoMiniSat.

Exercise 6

```
(setq formula3 '(((a) b) ((b) c) ((c) d) ((d) (a)) (a a)))
(to-dimacs formula3)
```

p cnf 4 5 -4 1 0 -1 2 0 -2 3 0 -3 -4 0 4 4 0

This was not satisfiable via CryptoMiniSat.

Exercise 8 A clique has k(k-1)/2 edges. So k=5 has 10 edges.

There are (k-1)!/2 distinct Hamiltonian cycles in a complete graph. A clique is a complete subgraph g of graph G, so therefore a clique with size k=5 will have:

```
(5-1)!/2 = 4!/2 = 4*3*2*1/2 = 24/2 = 12
```

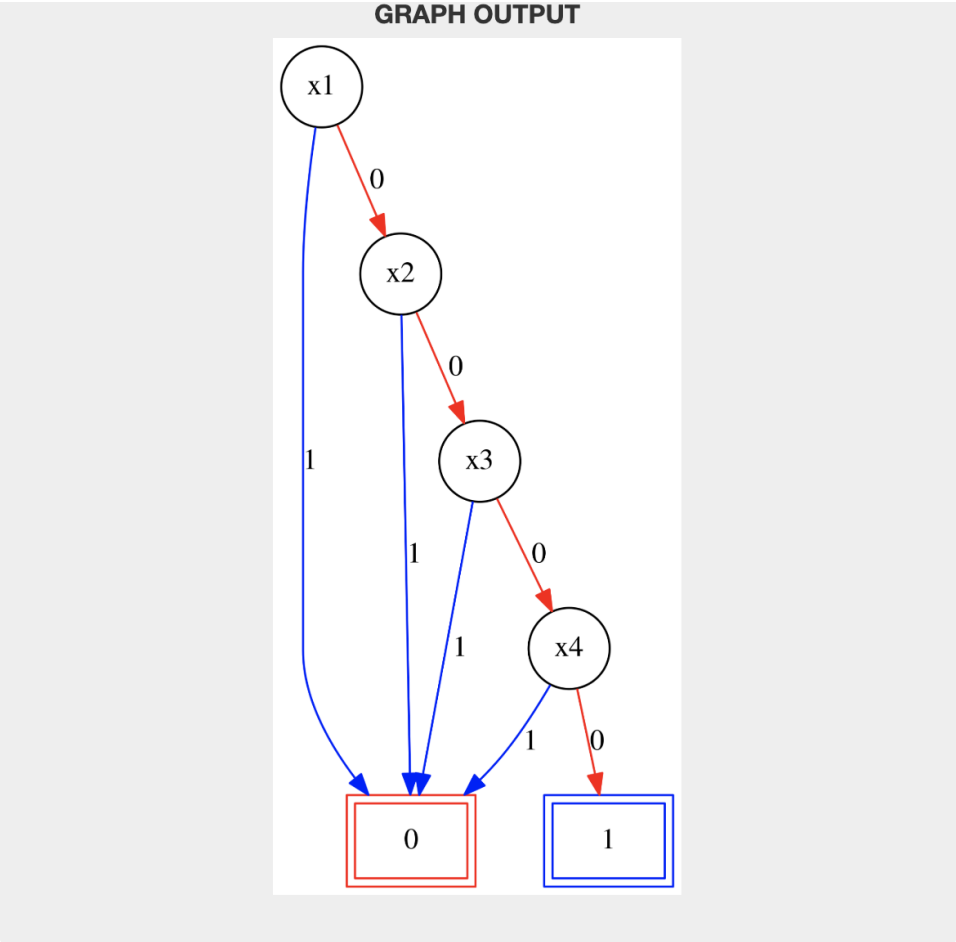
There will be (n-1)!/2 Hamiltonian cycles in a clique of size k=n

Week 12

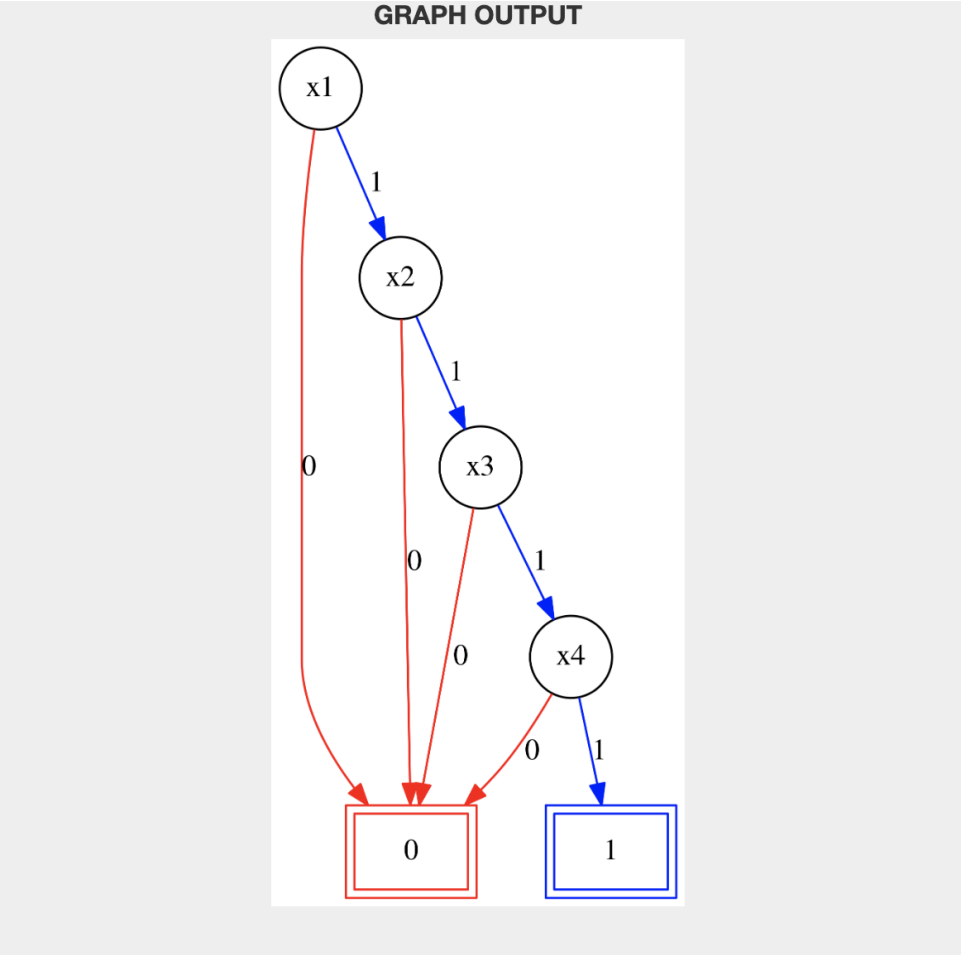
Chapter 17 Exercises

1

- 4 input NOR function



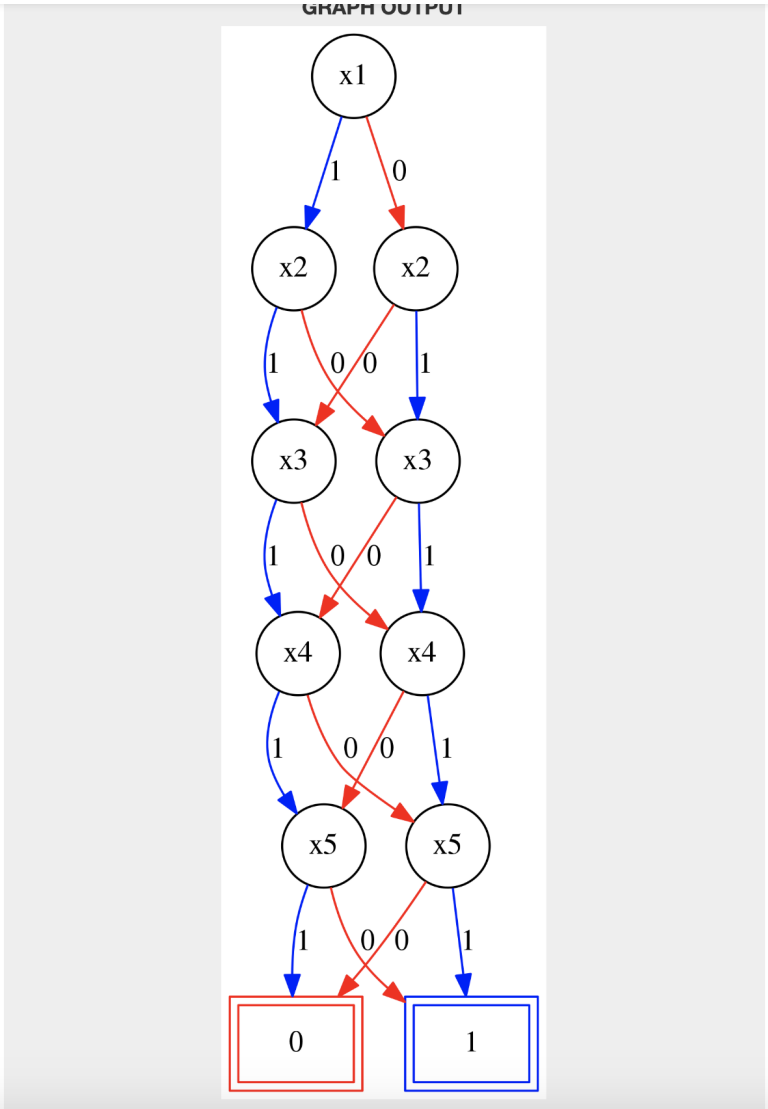
- 4 input AND function



- Comparison The have the same tree structure, but the edge labels are switched. Every edge labeled with a *1* in the 4 input NOR function is labeled with a *0* on the 4 input AND function and vice versa. Every edge labeled *0* in the 4 input NOR function is labeled with a *1* on the 4 input AND function.

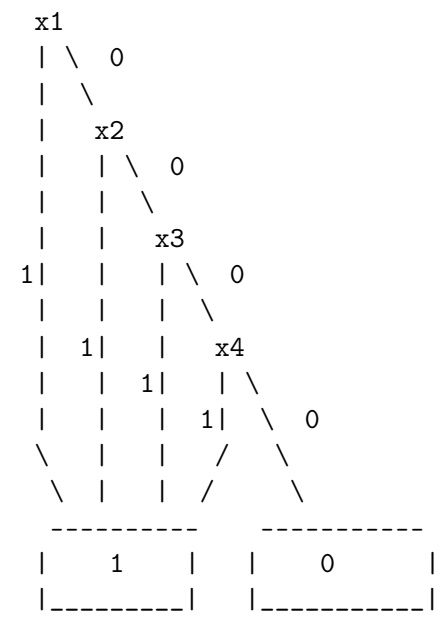
2

- 5 XNOR



- Comparison The only change is on the edges from e to the accept/reject states. The XNOR function swaps the 1 and 0 on each e node to the accept/reject state. This negates the output, which is the definition of the XNOR function.

3

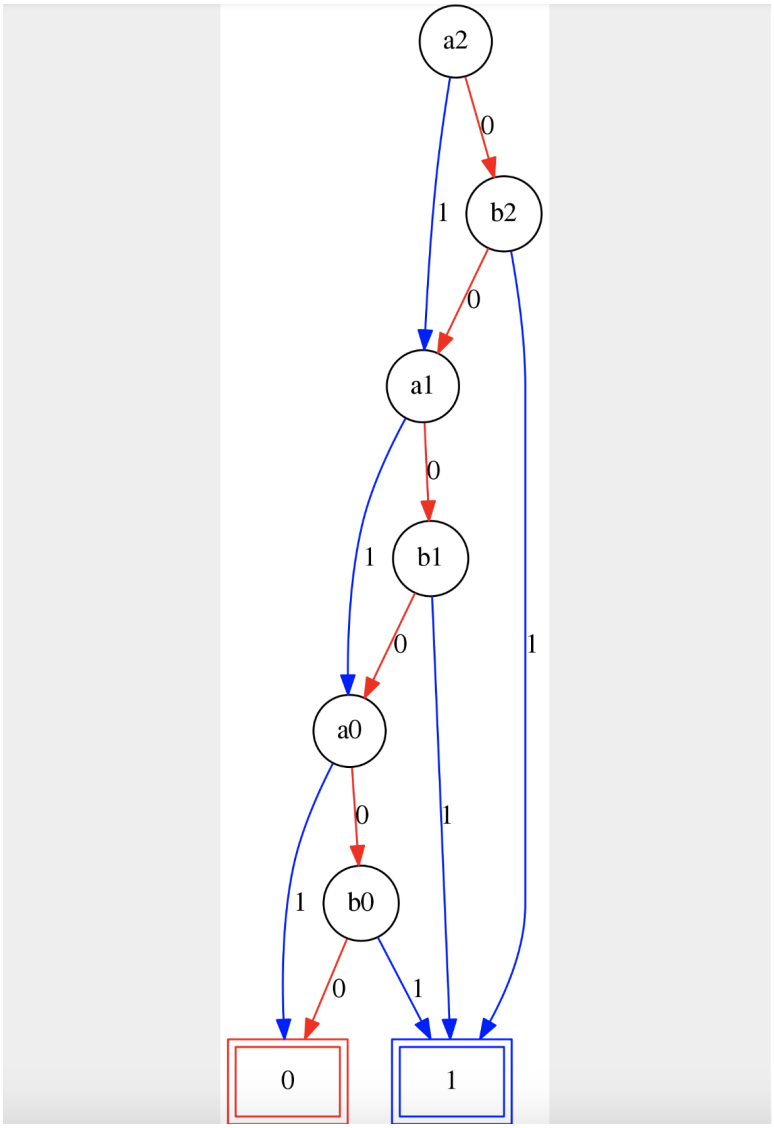


Yes, this is a linearly sized BDD. It is the same as the PBDD output

4

- a) No, it is incorrect. The BDD allows several invalid acceptance states such as the following where A > B:

  A: '010', B: '001'
  A: '100', B: '001'
  A: '101', B: '011'
  A: '110', B: '011'
  A: '110', B: '101'
  A: '100', B: '011'
  A: '101', B: '010'
  A: '110', B: '001'
  A: '100', B: '010'



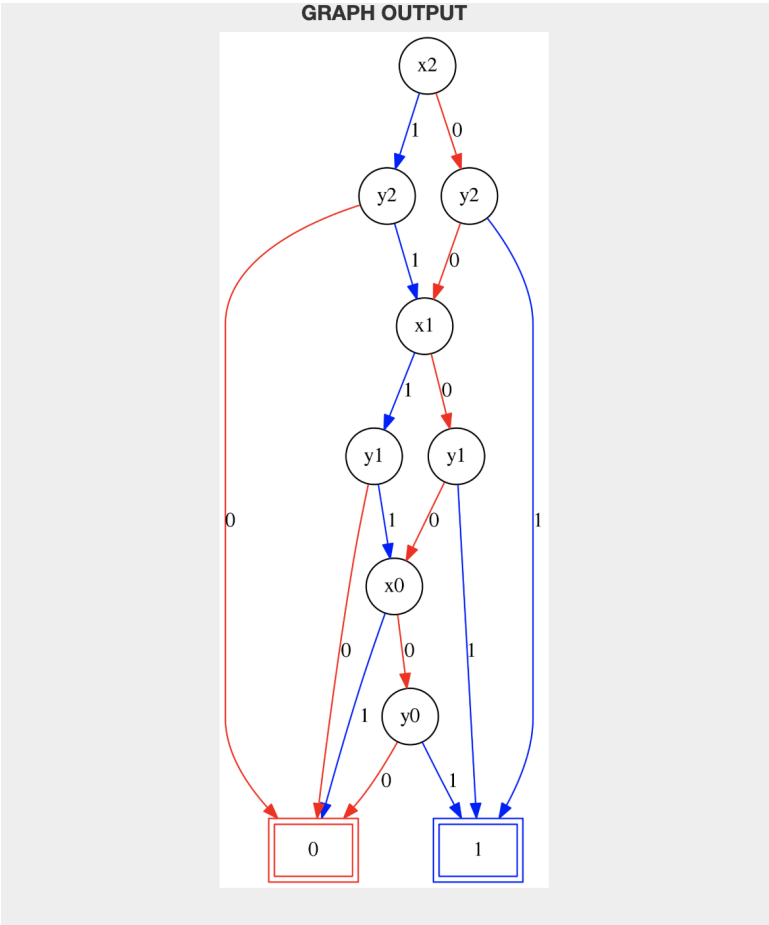
- b) Old

```
Var_Order : a2, b2, a1, b1, a0, b0
Main_Exp : ~a2 & b2 | ~a1 & b1 | ~a0 & b0
```

Modified

```
Var_Order : x2 y2 x1 y1 x0 y0
```

```
Main_Exp :    (~x2 & y2) |
              (~(x2 XOR y2) & (~x1 & y1)) |
              ((~(x2 XOR y2) & ~(x1 XOR y1)) & (~x0 & y0))
```



Week 13

Lambdafication

```
(fset 'I (set 'I (lambda (x) x)))

(fset 'Z (set 'Z (lambda (x) (lambda (y) y))))

(fset 'S
      (set 'S
```

```
             (lambda (a)
               (lambda (b)
                 (lambda (c)
                   (funcall b
                            (funcall
                               (funcall a b) c)))))))

(fset 'ADD
      (set 'ADD
           (lambda (a)
             (lambda (b) (funcall (funcall a S) b)))))

(fset 'MUL
      (set 'MUL
           (lambda (a)
             (lambda (b)
               (lambda (c)
                  (funcall a (funcall b c)))))))

(fset 'TRUE
      (set 'TRUE
           (lambda (a)
             (lambda (b)
                     a))))

(fset 'FALSE
      (set 'FALSE
           (lambda (a)
             (lambda (b)
               b))))

(fset 'NOT
      (set 'NOT
           (lambda (a)
             (funcall (funcall a FALSE) TRUE))))

(fset 'AND
      (set 'AND
           (lambda (a)
              (lambda (b)
```

```
                (funcall (funcall a b) a)))))


(fset 'OR
      (set 'OR
           (lambda (a)
             (lambda (b)
               (funcall (funcall a a) b)))))

(fset 'ZP
      (set 'ZP
          (lambda (a)
                  (funcall (funcall (funcall a FALSE) NOT) FALSE))))

(fset 'PAIR
     (set 'PAIR
          (lambda (x)
              (lambda (y)
                  (lambda (f)
                     (funcall (funcall f x) y))))))

(fset 'FIRST
     (set 'FIRST
          (lambda (p) (funcall p TRUE))))

(fset 'SECOND
     (set 'SECOND
          (lambda (p) (funcall p FALSE))))

(fset 'Y
     (set 'Y
        (lambda (f)
           ((lambda (x) (funcall x x)
            (lambda (x) (funcall f (lambda (&rest v)
                                     (apply (funcall x x) v)))))))))



(defun lambda-to-bool (b)
   (funcall (funcall b t) nil))
```

```
(defun bool-to-lambda (b)
   (if b
       TRUE
     FALSE))


(defun church-to-nat (c)
(funcall (funcall c '1+) 0))

(defun nat-to-church (n)
   (if (zerop n)
      Z
      (funcall S (nat-to-church (1- n)))))

(defun lambda-s (x)
   (1+ x))

(defun lambda-add (x y)
   (if (zerop y)
      x
   (lambda-s (lambda-add x (1- y)))))

(defun lambda-mul (x y)
   (if (zerop y)
       0
     (if (= 1 y)
        x
       (lambda-add x (lambda-mul x (1- y))))))

(defun lambda-ex (x y)
   (if (zerop y)
      1
    (if (= 1 y)
        x
      (lambda-mul x (lambda-ex x (1- y))))))

(lambda-to-bool (funcall (funcall AND TRUE) FALSE))
(bool-to-lambda nil)
```

Chapter 18 Exercises

Exercise 18.4.1 ADD applies the successor function repeatedly until the numbers have been added. MUL does the same, although instead of repeating the successor function, it repeats the addition function.

Exercise 18.4.2

```
(church-to-nat (funcall (funcall MUL (nat-to-church 4)) (nat-to-church 8)))
```

Exercise 18.5.1

```
(lambda-to-bool (funcall (funcall (funcall PAIR TRUE) FALSE) AND))
```

t

Exercise 18.5.2

```
(lambda-to-bool (funcall (funcall (funcall PAIR TRUE) FALSE) OR))
```

t

Exercise 18.8.1 A fixpoint combinator is a combinator E such that for any lambda expression G, the identity $(EG) = G(/(EG)$ holds. There are an infinite number of fixpoint combinators.

$Y = (\lambda\ \mathrm{f}.(\lambda\ \mathrm{x}.\ \mathrm{f}(\mathrm{x}\ \mathrm{x}))(\lambda\ \mathrm{x}.\ \mathrm{f}(\mathrm{x}\ \mathrm{x})))$

$Y_e = (\lambda\ f.(\lambda\ x.(x\ x))(\lambda\ y.f(\lambda\ v.\ ((y\ y)\ v))))$

$EG = (\lambda\ f.(\lambda\ x.\ (x\ x))(\lambda\ y.f(\lambda\ v.\ ((y\ y)\ v))))G$

$= (\lambda\ f.(\lambda\ x.x\ x)\ (\lambda\ y.f\ (\lambda\ v.\ y\ y\ v)))G$

$= (\lambda\ f.(\lambda\ y.f\ (\lambda\ v.y\ y\ v))(\lambda\ y.f\ (\lambda\ v.\ y\ y\ v)))G$

$= (\lambda\ y.G(\lambda\ v.\ y\ y\ v))(\lambda\ y.G(\lambda\ v.\ y\ y\ v\ ))$

$= G(\lambda\ v.\ (\lambda\ y.G(\lambda\ v.\ y\ y\ v\ ))(\lambda\ y.G(\lambda\ v.\ y\ y\ v\ ))\ v)$

$= G\ (\lambda\ y.G(\lambda\ v.\ y\ y\ v))(\lambda\ y.G(\lambda\ v.\ y\ y\ v))$

thus $EG = G(EG)$ and Y_e is a fixpoint combinator

Exercise 18.8.2

```
# Below, for clarity, we use don't use Church numerals..
# The Ye -- eager Y combinator

Ye = lambda f: (lambda x: x(x))(lambda y: f(lambda v: y(y)(v)))
# Pre-Factorial: performs the product of
```

```
# a natural number and all natural number less than it
# We call it pre-factorial because we need to apply
# Y to it to obtain the real factorial

prefact = lambda fact: lambda n: (1 if n==0 else n*fact(n-1))

# Pre-sum: sums all the natural numbers less than the given number

presum = lambda f: lambda n: (0 if n==0 else n+f(n-1))

# Pre-Fib: returns the nth number of the series defined by
# the following definitions
#  the first two numbers are 1 and 1
#  the next number is defined as the sum of the prior two numbers

prefib = lambda f: lambda n: 0 if n == 0 else (1 if n == 1 else f(n-1) + f(n-2))

fact = lambda n: Ye(prefact)(n)
sum = lambda n: Ye(presum)(n)

print("fact examples")
print(fact(3))
print(fact(5))
print(fact(8))
print("Sum examples")
print(sum(3))
print(sum(5))
print(sum(8))
```

Exercise 18.8.3 *Y_e is a fixed-point function due to Tennent's correspondence principle. Breaking down Y_e, we see that it is very similar to Y. The difference being that instead of evaluating the second λ x. f(x x) immedeiatly, we defer it into another wrapped lambda. This is essentially a no-op, as all it does it place the order of the inner function further back in queue for evalutation. This is why the applicative (eager) Y combinator works with eager languages. Instead of returning a "value" it returns another layered function. This means that eager languages will stop evaluation there, preventing an infinite loop of self evalutation.*