Solutions to selected exercises

Chapter 1

Exercise 1

- **a** General knowledge of poker could include the rules of poker, facts about a poker deck, probabilities of each kind of hand, and standard tendencies for human players (e.g., what types of bets people tend to make with certain kinds of hands).
- **b** One kind of prediction is what bet another player will make in the next round. For this prediction, the query is the player's next bet and the evidence consists of the cards in your hand, any open cards on the table, and any bets that have been made so far.
- **c** In poker, you would really like to figure out what cards other players have. These cards can be thought of as causes of the bets the other players have made. In this case, the query is another player's cards, and the evidence is the same as for part (b): your cards, the open cards on the table, and any bets that have been made so far.
- **d** Once you have figured out what cards another player has, you can use that information to predict any future bets they will make.

Chapter 2

Exercise 3

These programs produce different answers. In program (a), z is defined to be x, so x === z is always true. Therefore, this program produces 1.0. In program (b), z is defined to be y, so x === z is the same as asking whether x and y have the same value. There are two ways for x and y to have the same value. Either they are both true, which happens with probability 0.4 × 0.4 = 0.16, or they are both false, which happens with probability 0.6 × 0.6 = 0.36. Therefore, program (b) produces 0.16 + 0.36 = 0.52.

Exercise 4

```
    val die1 = FromRange(1, 7)
    val die2 = FromRange(1, 7)
    val total = Apply(die1, die2, (i1: Int, i2: Int) => i1 + i2)
    println(VariableElimination.probability(total, 11))
```

Exercise 5

```
    val die1 = FromRange(1, 7)
    val die2 = FromRange(1, 7)
    val total = Apply(die1, die2, (i1: Int, i2: Int) => i1 + i2)
    total.addCondition((i: Int) => i > 8)
    println(VariableElimination.probability(die1, 6))
```

Exercise 6

```
    def doubles = {
      val die1 = FromRange(1, 7)
      val die2 = FromRange(1, 7)
      die1 === die2
    }
    val jail = doubles && doubles && doubles
    println(VariableElimination.probability(jail, true))
```

Exercise 7

Here's a Figaro representation for this game:

```
    val numberOfSides =
        Select(0.2 -> 4, 0.2 -> 6, 0.2 -> 8, 0.2 -> 12, 0.2 -> 20)
    val roll = Chain(numberOfSides, ((i: Int) => FromRange(1, i + 1)))
```

Exercise 8

```
    val numberOfSides1 =
      Select(0.2 -> 4, 0.2 -> 6, 0.2 -> 8, 0.2 -> 12, 0.2 -> 20)
    val roll1 = Chain(numberOfSides1, ((i: Int) => FromRange(1, i + 1)))
    val numberOfSides2 =
      Select(0.2 -> 4, 0.2 -> 6, 0.2 -> 8, 0.2 -> 12, 0.2 -> 20)
    val roll2 = Chain(numberOfSides2, ((i: Int) => FromRange(1, i + 1)))
    def stickinessConstraint(sidesPair: (Int, Int)) =
      if (sidesPair._1 == sidesPair._2) 1.0 else 0.5
    val pairOfSides = ^^(numberOfSides1, numberOfSides2)
    pairOfSides.addConstraint(stickinessConstraint)
    println(VariableElimination.probability(roll2, 7))
// prints 0.05166666666666667
    roll1.observe(7)
    println(VariableElimination.probability(roll2, 7))
// prints 0.09704301075268815
```

Chapter 3

Exercise 2

Many answers are possible to this question. Here is one possible answer:

- **a** The security guard application has a set of sensors, such as cameras and touch sensors, that communicates the sensor readings to a central processor using a message bus. This central processor continually monitors inputs from the sensors and uses a probabilistic reasoning component to determine whether an anomalous situation is occurring. The central processor also contains a decision-making component that decides whether to raise an alert based on the findings of the probabilistic reasoning component. Finally, a user interface presents alerts, as well as current sensor readings, to an operator. In the context of this design, the inputs to the probabilistic anomaly detector at each point in time are the sensor readings, while the output is the probability of evidence, where a low probability of evidence indicates an anomaly.
- **b** Besides the sensor inputs, the variables in this application could include the locations and movements of people in the department store. You might also include variables to model the status of valuable items in the department store. Finally, you might include variables to capture the status of your sensors, for example, whether a camera is broken.
- **c** In this application, training data is plentiful because your security cameras can collect data continuously. The training data consists of sensor inputs taken over a long period of presumably normal activity. From this data, you can learn knowledge of typical normal activity, such as patterns of movement of people, locations of items, and common disruptions to sensor input. You probably wouldn't learn precisely what an anomaly looks like, because you won't have data of actual anomalies.

Exercise 3

Again, there are many possible answers to this question. Here is one possibility:

- **a** One set of variables is the cards held by each player. The probabilities of these variables are derived mathematically from the possible ways to deal cards. You don't have to learn these probabilities from data. Another set of variables is the bets made by each player. This includes both past bets that you observe, and from which you try to infer the cards, and future bets that you try to predict. Since these variables result from unpredictable human behavior, you will want to learn their probabilities from experience.
- **b** In this application, you might have two levels of knowledge. The first level includes knowledge about how people play poker in general. The second level is specific to a particular player and includes knowledge about how that particular player plays. Accordingly, there will be two learning components in

the architecture. One component will be player-specific, and will learn how that player tends to play. From the point of view of this component, the system's knowledge of how people play in general constitutes general knowledge that informs its beliefs about this particular player. In particular, the prior model for this component comes from your general knowledge of how people play. The second learning component is for learning this general knowledge. The data to learn the model for this component comes from the first component. After observing how a particular player plays, the system can update its beliefs about how people play in general. This kind of model, with learning happening on more general and more specific levels, is knows as a hierarchical model.

Chapter 4

Exercise 1

Each possible world specifies the card held by each player. There are twenty possible worlds; the first player can have any of the five cards and the other player can have any of the four remaining cards. Assuming the deck is fairly shuffled, each possible world has probability 1/20.

Exercise 2

When we observe evidence that the first player has a picture card, this rules out the possible worlds where the first player has the ace of spades. This leaves sixteen possible worlds. In ten of them, the second player has a spade, so the probability the second player has a spade given that the first player has a picture card is 10/16.

Exercise 3

- **a** The variables are the card of each player, player 1's first action, player 2's action, and player 1's second action, which only matters if player 1's first action is to bet and player 2's action is to bet.
- **b** The card held by player 1 does not depend on any other variables. Player 2's card depends on player 1's card, because they can't have the same card. Player 1's first action depends on her card. Player 2's action depends on his card and player 1's first action. Player 1's second action depends on her card, her first action, and player 2's action. Note that the player's actions don't depend directly on the opponent's cards because they are unknown to the player.
- **c** Player 1's card is chosen uniformly from the set of five cards. Player 2's card is chosen uniformly from the four cards remaining after player 1's card is removed from the deck. Each action of a player has a conditional probability distribution that specifies a dependency on the card held by the player and the previous actions. Given the values of those variables, the functional form for the action will be a `Flip` or `Select`, because only two choices are available for each action.

d The parameters of the variables representing the cards of the two players are known. The parameters of the variables representing the actions of the players need to be estimated or learned from experience.

Exercise 4

There are many ways to solve this problem. Here is my answer. Since this problem is more advanced, I'm showing the entire code. First the model:

```
import com.cra.figaro.library.atomic.discrete
import com.cra.figaro.language.Chain
import com.cra.figaro.library.compound.{RichCPD, OneOf, ×}
import com.cra.figaro.language.{Flip, Constant, Apply}
import com.cra.figaro.algorithm.factored.VariableElimination

object ex44 {
  def main(args: Array[String]) {
    // To keep the code simple, I just make the cards an integer
    val cards = List(5, 4, 3, 2, 1)
    // The discrete uniform distribution chooses uniformly from a fixed
    // set of possibilities
    val player1Card = discrete.Uniform(cards:_×)
    val player2Card =
      Chain(player1Card, (card: Int) =>
          // Player 2 can get any card except the first player's card
          discrete.Uniform(cards.filter(_ != card):_×))
    val player1Bet1 =
      RichCPD(player1Card,
          // Player 1 is more likely to bet with a higher card,
          // but will sometimes bet with a lower card to bluff
          OneOf(5, 4, 3) -> Flip(0.9),
          × -> Flip(0.4) // ×××Change this for part (c)×××
      )
    val player2Bet =
      RichCPD(player2Card, player1Bet1,
          (OneOf(5, 4), ×) -> Flip(0.9),
          (×, OneOf(false)) -> Flip(0.5),
          (×, ×) -> Flip(0.1))
    val player1Bet2 =
      Apply(player1Card, player1Bet1, player2Bet,
            (card: Int, bet11: Boolean, bet2: Boolean) =>
              // Player 1's second bet is only relevant if she passed the
              // first time and player 2 bet
                    !bet11 && bet2 && (card == 5 || card == 4))

    // This element represents the gain to player 1 from the game. I have
    // made it an Element[Double] so I can query its mean.
    val player1Gain =
      Apply(player1Card, player2Card, player1Bet1, player2Bet, player1Bet2,
          (card1: Int, card2: Int, bet11: Boolean,
           bet2: Boolean, bet12: Boolean) =>
            if (!bet11 && !bet2) 0.0
            else if (bet11 && !bet2) 1.0
            else if (!bet11 && bet2 && !bet12) -1.0
            else if (card1 > card2) 2.0
            else -2.0)
```

a. To make a decision, player 1 needs to consider her expected gain if she passes and expected gain if she bets. Here is code to make the necessary computations. It turns out that with the parameters I've used, both cases produce an expected gain of 0.25, so her decision doesn't matter in this case.

```
    player1Card.observe(4)
    player1Bet1.observe(true)
    val alg1 = VariableElimination(player1Gain)
    alg1.start()
    alg1.stop()
    println("Expected gain for betting:" + alg1.mean(player1Gain))
    player1Bet1.observe(false)
    val alg2 = VariableElimination(player1Gain)
    alg2.start()
    alg2.stop()
    println("Expected gain for passing:" + alg2.mean(player1Gain))
    player1Card.unobserve()
    player1Bet1.unobserve()
```

b. Here is the code for player 2's decision. The expected gain for player 1 if player 2 passes is 1.0 (since player 2 folds), while the expected gain if player 2 bets is about 0.77. Since player 1's expected gain is lower, player 2 should bet.

```
    player2Card.observe(3)
    player1Bet1.observe(true)
    player2Bet.observe(true)
    val alg3 = VariableElimination(player1Gain)
    alg3.start()
    alg3.stop()
    println("Expected gain for betting:" + alg3.mean(player1Gain))
    player2Bet.observe(false)
    val alg4 = VariableElimination(player1Gain)
    alg4.start()
    alg4.stop()
    println("Expected gain for passing:" + alg4.mean(player1Gain))

  }
}
```

c. If you change the parameter in the line marked "Change this for part (c)" to 0.01, you will virtually eliminate the possibility that player 1 bluffs. In this case, the expected gain to player 1 if player 2 bets is very high, because player 1 has a higher card, so player 2 should fold.

Chapter 5

Exercise 1

a. Directed from player's cards to player's bet

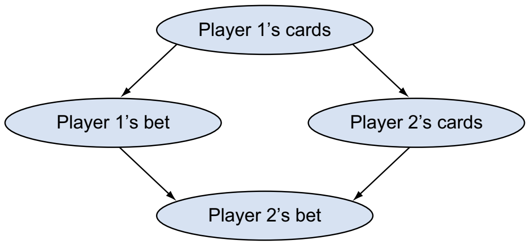
b. This could be modeled in different ways. You could select player 1's card and then select player 2's card from the remaining cards. In this case, the dependency would be from player 1's card to player 2's card. You could also reverse the order of the two players. Finally, you could model each of the players' cards as being drawn from the complete deck and add an undirected dependency between them to represent the fact that they can't be the same.

- **c** Today's weather might influence my mood, but not the other way round.
- **d** Different answers are possible. Your mood might determine whether you eat breakfast. Eating breakfast might also improve your mood. A correct model might have two separate mood variables, one for before breakfast time and one for after breakfast time. Your mood before breakfast time might influence whether or not you eat breakfast, and both these variables will influence your mood after breakfast.
- **e** Ordinarily, the temperature in your living room depends on the thermostat setting. The higher the thermostat setting, the warmer your living room will be. This model is adequate. However, you could make an argument that if your living room is cold, you will turn the thermostat up. This creates a feedback loop between the thermostat setting, the temperature in your living room, and your action. Feedback loops are best represented as dynamic models (see chapter 8).
- **f** The thermometer reading is a sensor of the temperature, so it depends probabilistically on the temperature.
- **g** The temperatures in the living room and in the kitchen are related by an undirected relationship.
- **h** The topic of a news article determines its content, so the content depends on the topic.
- **i** Typically, a writer will generate the summary of an article after reading the article. If this is the case, the summary depends on the content.

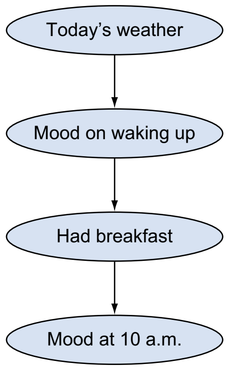
Exercise 2

The following Bayesian networks are one possible answer to each question. Arguments could be made for other networks. The important thing is to make a considered design choice.

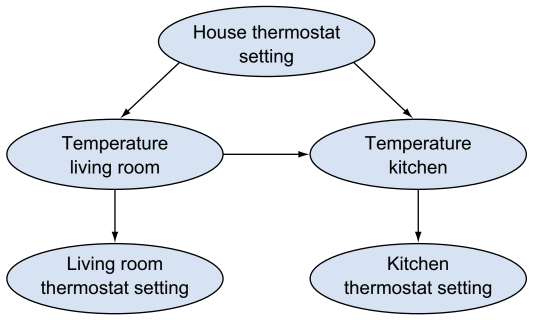
a



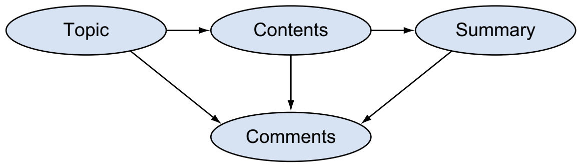
b



c This one's tricky since the relationship between the temperature in the kitchen and in the living room is best modeled by an undirected dependency, which can't be done in a Bayesian network. Therefore, if you want to use a Bayesian network, you'll need to use a directed dependency, which could go either way.



d



Exercise 6

The solution to this exercise also includes solutions to exercises 4 and 5.

```
def tennis(
    probP1ServeWin: Double, probP1Winner: Double, probP1Error: Double,
    probP2ServeWin: Double, probP2Winner: Double, probP2Error: Double):
  Element[Boolean] = {
  def rally(firstShot: Boolean, player1: Boolean): Element[Boolean] = {
    val pWinner =
      if (firstShot && player1) probP1ServeWin
      else if (firstShot && !player1) probP2ServeWin
      else if (player1) probP1Winner
      else probP2Winner
    val pError = if (player1) probP1Error else probP2Error
    val winner = Flip(pWinner)
    val error = Flip(pError)
    If(winner, Constant(player1),
       If(error, Constant(!player1),
          rally(false, !player1)))
  }

  def game(
      p1Serves: Boolean, p1Points: Element[Int],
      p2Points: Element[Int]): Element[Boolean] = {
    val p1WinsPoint = rally(true, p1Serves)
    val newP1Points =
      Apply(p1WinsPoint, p1Points, (wins: Boolean, points: Int) =>
        if (wins) points + 1 else points)
    val newP2Points =
      Apply(p1WinsPoint, p2Points, (wins: Boolean, points: Int) =>
        if (wins) points else points + 1)
    val p1WinsGame =
      Apply(newP1Points, newP2Points, (p1: Int, p2: Int) =>
        p1 >= 4 && p1 - p2 >= 2)
    val p2WinsGame =
      Apply(newP2Points, newP1Points, (p2: Int, p1: Int) =>
        p2 >= 4 && p2 - p1 >= 2)
    val gameOver = p1WinsGame || p2WinsGame
    If(gameOver, p1WinsGame, game(p1Serves, newP1Points, newP2Points))
  }

  def play(
      p1Serves: Boolean, p1Sets: Element[Int], p2Sets: Element[Int],
      p1Games: Element[Int], p2Games: Element[Int]): Element[Boolean] = {
    val p1WinsGame = game(p1Serves, Constant(0), Constant(0))
    val newP1Games =
      Apply(p1WinsGame, p1Games, p2Games,
            (wins: Boolean, p1: Int, p2: Int) =>
                if (wins) {
                  if (p1 >= 5) 0 else p1 + 1
                } else {
                  if (p2 >= 5) 0 else p1
                })
        val newP2Games =
          Apply(p1WinsGame, p1Games, p2Games,
                (wins: Boolean, p1: Int, p2: Int) =>
                    if (wins) {
                      if (p1 >= 5) 0 else p2
                    } else {
                      if (p2 >= 5) 0 else p2 + 1
                    })
        val newP1Sets =
          Apply(p1WinsGame, p1Games, p1Sets,
                (wins: Boolean, games: Int, sets: Int) =>
                  if (wins && games == 5) sets + 1 else sets)
        val newP2Sets =
          Apply(p1WinsGame, p2Games, p2Sets,
                (wins: Boolean, games: Int, sets: Int) =>
                  if (!wins && games == 5) sets + 1 else sets)
        val matchOver =
          Apply(newP1Sets, newP2Sets, (p1: Int, p2: Int) =>
            p1 >= 2 || p2 >= 2)
        If(matchOver,
           Apply(newP1Sets, (sets: Int) => sets >= 2),
           play(!p1Serves, newP1Sets, newP2Sets, newP1Games, newP2Games))
      }
      play(true, Constant(0), Constant(0), Constant(0), Constant(0))
    }
```

Chapter 6

Exercise 1

```
val teacherSkill = Array.fill(6)(Flip(0.6))
val studentAbility = Array.fill(6, 30)(Flip(0.7))
val pass = Array.tabulate(6, 30)((cls: Int, stdnt: Int) =>
  CPD(teacherSkill(cls), studentAbility(cls)(stdnt),
      (true, true) -> Flip(0.9),
      (true, false) -> Flip(0.6),
      (false, true) -> Flip(0.7),
      (false, false) -> Flip(0.3)))

for {
  cls <- 0 until 6
  stdnt <- 0 until 30
} {
  pass(cls)(stdnt).observe(data(cls)(stdnt) == 'P')
}

val ve =
  VariableElimination(teacherSkill(0), teacherSkill(1), teacherSkill(2),
                      teacherSkill(3), teacherSkill(4), teacherSkill(5))
ve.start()
for { cls <- 0 until 6 } {
  println("Teacher " + cls + ": " + ve.probability(teacherSkill(cls),
 true))
}
```

Exercise 4

```
val numRows = 10
val numColumns = 10
val mine = Array.fill(numRows, numColumns)(Select(0.4 -> 1, 0.6 -> 0))
val count = Array.tabulate(numRows, numColumns)((i: Int, j: Int) => {
  val neighbors =
    for {
      k <- i - 1 to i + 1
      l <- j - 1 to j + 1
      if k >= 0 && k < numRows
      if l >= 0 && l < numColumns
      if k != i || l != j
    } yield mine(k)(l)
  Container(neighbors:_*).reduce(_ + _)
})

val data =
  Array("?2M???????", "1?????????", "?1????????", "??????????",
        "??????????", "??????????", "??????????", "??????????",
        "??????????", "??????????")
for {
  i <- 0 until numRows
  j <- 0 until numColumns
} {
  data(i)(j) match {
    case 'M' => mine(i)(j).observe(1)
    case d if d.isDigit =>
      mine(i)(j).observe(0)
      count(i)(j).observe(d - '0')
    case '?' => ()
  }
}

println(VariableElimination.probability(mine(1)(1), 1))
```

Exercise 5

```
val purchases =
  VariableSizeArray(Binomial(100, 0.1), (i: Int) =>
    If(Flip(0.5), FromRange(1, 11), Constant(0)))
val total = purchases.foldLeft(0)(_ + _).map(_.toDouble)
val alg = Importance(10000, total)
alg.start()
println(alg.mean(total))
```

Exercise 6

The following solution combines Scala collections with Figaro collections. A subtlety that arises is that the groups must not be created inside a chain; otherwise, the constraints won't be applied properly. Therefore, in this solution, the groups are "top-level" objects.

```
class Group extends Container[Int, Int] {
  val indices = 0 until 5
```

```
    val numChildren = FromRange(1, 6)

    def generate(i :Int) =
      Chain(numChildren, (n: Int) =>
        if (n > i) If(Flip(0.5), FromRange(1, 11), Constant(0))
        else Constant(0))

    def generate(indices: List[Int]): Map[Int, Element[Int]] = {
      val result = indices.map(i => (i, apply(i))).toMap
      for {
        i <- indices
        j <- indices
        if j > i
      } {
        val pairElem = ^^(result(i), result(j))
        pairElem.addConstraint((pair: (Int, Int)) =>
          if ((pair._1 > 0) && (pair._2 > 0)) 1.0 else 0.8)
      }
      result
    }

    // ensure that the constraints are created as soon as the group is
     created
    generate(indices.toList)
  }

  def main(args: Array[String]) {
    val groups = Array.fill(40)(new Group)
    val groupExists = Array.fill(40)(Flip(0.1))
    val groupTotals =
      for { i <- 0 until 40 } yield {
        If(groupExists(i),
           groups(i).foldLeft(0)(_ + _),
           Constant(0))
      }
    val grandTotal: Element[Double] =
      Container(groupTotals:_*).foldLeft(0)(_ + _).map(_.toDouble)
    val alg = Importance(10000, grandTotal)
    alg.start()
    println(alg.mean(grandTotal))

  }
```

Chapter 7

Exercise 1

```
  class Actor {
    val liked = Uniform(0, 1)
  }

  class Movie(actors: List[Actor]) {
    private val likedContainer = Container(actors.map(_.liked):_*)
    private val popularityMean = likedContainer.foldLeft(0.0)(_ max _)
    val popularity = Normal(popularityMean, 0.01)
  }
    val actors = Array.fill(10)(new Actor)
    val movie1 = new Movie(List(actors(0), actors(1)))
    val movie2 = new Movie(List(actors(0), actors(2), actors(3)))
    val movie3 = new Movie(List(actors(1), actors(4), actors(5)))
    val movie4 = new Movie(List(actors(2), actors(4), actors(6)))
    val movie5 =
      new Movie(List(actors(5), actors(7), actors(8), actors(9)))
    val movie6 = new Movie(List(actors(6), actors(7), actors(8)))
    val movie7 = new Movie(List(actors(1), actors(4), actors(9)))

    // The evidence is provided as soft constraints. Hard conditions would
    // be extremely unlikely to be satisfied during sampling
    def constrain(movie: Movie, pop: Double) {
      movie.popularity.addConstraint(d =>
        math.exp(-(pop-d)×(pop-d)/0.01))
    }
    constrain(movie1, 0.4)
    constrain(movie2, 0.2)
    constrain(movie3, 0.7)
    constrain(movie4, 0.5)
    constrain(movie5, 0.8)
    constrain(movie6, 0.4)

    val alg =
      MetropolisHastings(100000, ProposalScheme.default, movie7.popularity)
    alg.start()
    println(alg.mean(movie7.popularity))
```

Exercise 3

```
class Student {
  val ability = Uniform(0, 1)
}

class Subject {
  val difficulty = Uniform(0, 1)
}

class Instructor {
  val quality = Uniform(0, 1)
}

case class Course(subject: Subject, instructor: Instructor)

case class Enrollment(student: Student, course: Course) {
  val gradeMax =
    Apply(student.ability, course.instructor.quality,
          course.subject.difficulty,
          (ability: Double, quality: Double, difficulty: Double) =>
            (ability + quality - difficulty).max(0.0).min(1.0))
  val grade = Uniform(Constant(0.0), gradeMax)
}

val students = Array.fill(5)(new Student)
val subjects = Array.fill(3)(new Subject)
val instructors = Array.fill(2)(new Instructor)
val course1 = Course(subjects(0), instructors(0))
val course2 = Course(subjects(1), instructors(0))
val course3 = Course(subjects(1), instructors(1))
val course4 = Course(subjects(2), instructors(1))
val enrollment1 = Enrollment(students(0), course1)
val enrollment2 = Enrollment(students(0), course2)
val enrollment3 = Enrollment(students(0), course3)
val enrollment4 = Enrollment(students(1), course1)
val enrollment5 = Enrollment(students(1), course2)
val enrollment6 = Enrollment(students(2), course2)
val enrollment7 = Enrollment(students(2), course3)
val enrollment8 = Enrollment(students(2), course4)
val enrollment9 = Enrollment(students(3), course1)
val enrollment10 = Enrollment(students(3), course3)
val enrollment11 = Enrollment(students(3), course4)
val enrollment12 = Enrollment(students(4), course4)

def constrain(enrollment: Enrollment, grd: Double) {
  enrollment.grade.addConstraint(d => math.exp(-(grd-d)×(grd-d)/0.01))
}
constrain(enrollment1, 0.3)
constrain(enrollment2, 0.6)
constrain(enrollment3, 0.4)
constrain(enrollment4, 0.5)
constrain(enrollment5, 0.8)
constrain(enrollment6, 0.3)
constrain(enrollment7, 0.2)
constrain(enrollment8, 0.2)
constrain(enrollment9, 0.7)
constrain(enrollment10, 0.1)
constrain(enrollment11, 0.2)
constrain(enrollment12, 0.1)

val alg =
  MetropolisHastings(100000, ProposalScheme.default,
    students(1).ability, subjects(2).difficulty, instructors(1).quality)
alg.start()
println("students(1).ability: " + alg.mean(students(1).ability))
println("subjects(2).difficulty: " + alg.mean(subjects(2).difficulty))
println("instructors(1).quality: " + alg.mean(instructors(1).quality))
```

Exercise 4

Add the following lines to the code for the previous exercise.

```
val newCourse1 = Course(subjects(2), instructors(0))
val newCourse2 = Course(subjects(2), instructors(1))
val newEnrollment1 = Enrollment(students(0), newCourse1)
val newEnrollment2 = Enrollment(students(0), newCourse2)
val newEnrollment = Select(0.5 -> newEnrollment1, 0.5 -> newEnrollment2)
val predictedGrade = Chain(newEnrollment, (enrollment: Enrollment) =>
  enrollment.grade)
```

Chapter 8

Exercise 1

```
abstract class State {
  val alicePoints: Element[Int]
  val bobPoints: Element[Int]
  // Note: terminal and bobWinning are defs to avoid null pointer
  // exceptions on initialization. It's a Scala quirk that when you
  // initialize a variable that uses abstract variables in it's
  // definition, it will throw a null pointer exception.
  def terminal: Element[Boolean] =
    Apply(alicePoints, bobPoints, (a: Int, b: Int) => a >= 21 || b >= 21)
  def bobWinning: Element[Boolean] =
    Apply(alicePoints, bobPoints, (a: Int, b: Int) => b > a)
}

case class InitialState() extends State {
  val alicePoints = Constant(0)
  val bobPoints = Constant(0)
}

case class NextState(current: State) extends State {
  val aliceWinsPoint = Flip(0.52)
  val alicePoints =
    Apply(current.alicePoints, aliceWinsPoint,
          (curr: Int, wins: Boolean) => if (wins) curr + 1 else curr)
  val bobPoints =
    Apply(current.bobPoints, aliceWinsPoint,
          (curr: Int, wins: Boolean) => if (!wins) curr + 1 else curr)
}

def playFrom(current: State): Element[Boolean] = {
  If(current.terminal, current.bobWinning, playFrom(NextState(current)))
}

val bobWins = playFrom(InitialState())

println(Importance.probability(bobWins, true))
```

Exercise 2

Add the following line at the beginning of the program:

```
val probAliceWinsPoint = Beta(2, 2)
```

Change the definition of `aliceWinsPoint` to

```
val aliceWinsPoint = Flip(probAliceWinsPoint)
```

Add the following lines to observe that the score after 19 points is 11-8 in favor of Alice and to continue play from that point on:

```
def nthState(n: Int): State = {
  if (n <= 0) InitialState() else NextState(nthState(n-1))
}

val nineteenthState = nthState(19)
nineteenthState.alicePoints.observe(11)
nineteenthState.bobPoints.observe(8)
val bobWins = playFrom(nineteenthState)
```

Exercise 5

```
abstract class Network {
  val nodes: Element[List[Int]]
  // We use an immutable map to represent the state of edges in a
  // network. If an edge is added or deleted, a new map is created that
  // is changed in one place.
  val edges: Element[Map[(Int, Int), Boolean]]
}

case class InitialNetwork() extends Network {
  // start with two nodes so we can add an edge on the first iteration
  val nodes = Constant(List(1, 0))
  val edges = Constant(Map[(Int, Int), Boolean]())
}

case class NextNetwork(current: Network) extends Network {
  val newNodeAdded = Flip(0.1)
  val nodes =
    Apply(current.nodes, newNodeAdded, (ns: List[Int], b: Boolean) =>
      if (b) ns.length :: ns else ns)
  val edges =
    If(newNodeAdded,
       current.edges,
       Chain(nodes, (ns: List[Int]) => {
         val first = discrete.Uniform(ns:_*)
         val second = discrete.Uniform(ns.filterNot(_ == first):_*)
         Apply(current.edges, first, second,
               (es: Map[(Int, Int), Boolean], f: Int, s: Int) =>
                  es + ((f, s) -> !es.getOrElse((f, s), false)))
       }))
}

def nthNetwork(n: Int): Network =
  if (n <= 0) InitialNetwork() else NextNetwork(nthNetwork(n - 1))

val count =
  Apply(nthNetwork(100).edges, (m: Map[(Int, Int), Boolean]) =>
    m.values.filter(_ == true).size.toDouble)
val alg = Importance(10000, count)
alg.start()
println(alg.mean(count))
```

Chapter 9

Exercise 1

In the following solutions, I use the abbreviation PPBO for Printer Power Button On, TL for Toner Level, TLLI for Toner Level Low Indicator, PF for Paper Flow, PJIO for Paper Jam Indicator On, and PS for Printer State.

a P(PPBO = true) × P(TL = low | PPBO = true) × P(TLLI = false | PPBO = true, TL = low) ×
P(PF = smooth | PPBO = true, TL = low, TLLI = false) ×
P(PJIO = false | PPBO = true, TL = low, TLLI = false, PF = smooth) ×
P(PS = poor | PPBO = true, TL = low, TLLI = false, PF = smooth, PJIO = false)

b P(PPBO = true) × P(TL = low) × P(TLLI = false | TL = low) × P(PF = smooth) ×
P(PJIO = false | PF = smooth) × P(PS = poor | PPBO = true, TL = low, PS = poor)

c P(PPBO) × P(TL) × P(TLLI | TL) × P(PF) × P(PJIO | PF) × P(PS | PPBO, TL, PS)

Exercise 2

a P(PPBO = true) =

$$\Sigma_{tl}\,\Sigma_{tlli}\,\Sigma_{pf}\,\Sigma_{pjio}\,\Sigma_{ps}\; P(\text{PPBO} = \text{true}) \times P(\text{TL} = \text{tl}) \times P(\text{TLLI} = \text{tlli}) \times P(\text{PF} = \text{pf}) \times P(\text{PJIO} = \text{pjio} \mid \text{PF} = \text{pf}) \times P(\text{PS} = \text{ps} \mid \text{PPBO} = \text{true}, \text{TL} = \text{tl}, \text{PF} = \text{pf})$$

b P(PPBO = true) =

$$\Sigma_{tl}\,\Sigma_{tlli}\,\Sigma_{pf}\,\Sigma_{pjio}\; P(\text{PPBO} = \text{true}) \times P(\text{TL} = \text{tl}) \times P(\text{TLLI} = \text{tlli}) \times P(\text{PF} = \text{pf}) \times P(\text{PJIO} = \text{pjio} \mid \text{PF} = \text{pf}) \times P(\text{PS} = \text{poor} \mid \text{PPBO} = \text{true}, \text{TL} = \text{tl}, \text{PF} = \text{pf})$$

Exercise 3

a
$$P(POTUS) = \frac{1}{40{,}000{,}000}$$

$$P(LH|POTUS) = \frac{1}{3}$$

$$P(LH|!POTUS) = \frac{1}{10}$$

By Bayes' rule

$$P(POTUS|LH) = \frac{P(POTUS)P(LH|POTUS)}{P(POTUS)P(LH|POTUS) + P(!POTUS)P(LH|!POTUS)}$$

$$= \frac{40{,}000{,}000 \cdot 2}{\frac{1}{40{,}000{,}000} \times \frac{1}{2} + \frac{39{,}999{,}999}{40{,}000{,}000} \times \frac{1}{10}} \approx \frac{1}{2{,}000{,}000}$$

Becoming president is five times more likely than before but still extremely rare.

b $P(HU|POTUS) = \dfrac{3}{20}$

$$P(LH|!POTUS) = \frac{1}{2{,}000}$$

$$P(POTUS|HU) = \frac{P(POTUS)P(HU|POTUS)}{P(POTUS)P(HU|POTUS) + P(!POTUS)P(HU|!POTUS)}$$

$$= \frac{\frac{1}{40{,}000{,}000} \cdot \frac{3}{20}}{\frac{1}{40{,}000{,}000} \times \frac{3}{20} + \frac{39{,}999{,}999}{40{,}000{,}000} \times \frac{1}{10}} \approx \frac{1}{2{,}666{,}667}$$

Note that although a much higher fraction of US presidents went to Harvard than the general population, it's still extremely unlikely that a random Harvard student will become president.

c Since left-handedness and going to Harvard are conditionally independent given whether someone became president,

$$P(LH,HU|POTUS) = P(LH|POTUS)P(HU|POTUS) = \frac{1}{2} \times \frac{3}{20} = \frac{3}{40}$$

$$P(LH,HU|!POTUS) = P(LH|!POTUS)P(HU|!POTUS)$$

$$= \frac{1}{10} \times \frac{1}{20000} = \frac{1}{20{,}000}$$

$$P(POTUS|LH,HU)$$

$$= \frac{P(POTUS)P(HU|POTUS)}{P(POTUS)P(LH,HU|POTUS) + P(!POTUS)P(LH,HU|!POTUS)}$$

$$= \frac{\frac{1}{40{,}000{,}000} \cdot \frac{3}{40}}{\frac{1}{40{,}000{,}000} \times \frac{3}{20} + \frac{39{,}999{,}999}{40{,}000{,}000} \times \frac{1}{10}} \approx \frac{1}{133{,}334}$$

As you add independent pieces of evidence, the rare event starts to become less rare.

Chapter 10

Exercise 1

P(Hungry)

Hungry	
False	3/4
True	1/4

P(Eat | Hungry)

Hungry	Eat	
False	None	2/3
False	A little	1/3
False	A lot	0
True	None	1/6
True	A little	1/3
True	A lot	1/2

P(Tired | Eat)

Eat	Tired	
None	False	3/4
None	True	1/4
A little	False	1/2
A little	True	1/2
A lot	False	1/4
A lot	True	3/4

P(Cry | Hungry, Tired)

Hungry	Tired	Cry	
False	False	False	5/6
False	False	True	1/6
False	True	False	1/6

(continued)

Hungry	Tired	Cry	
False	True	True	5/6
True	False	False	1/6
True	False	True	5/6
True	True	False	0
True	True	True	1

Exercise 2

P(Hungry, Eat, Tired, Cry) = P(Hungry) P(Eat | Hungry) P(Tired | Eat) P(Cry | Hungry, Tired) = Product of the four factors =

Hungry	Eat	Tired	Cry	
False	None	False	False	3/4×2/3×3/4×5/6 = 15/48
False	None	False	True	3/4×2/3×3/4×1/6 = 3/48
False	None	True	False	3/4×2/3×1/4×1/6 = 1/48
False	None	True	True	3/4×2/3×1/4×5/6 = 5/48
False	A little	False	False	3/4×1/3×1/2×5/6 = 5/48
False	A little	False	True	3/4×1/3×1/2×1/6 = 1/48
False	A little	True	False	3/4×1/3×1/2×1/6 = 1/48
False	A little	True	True	3/4×1/3×1/2×5/6 = 5/48
False	A lot	False	False	3/4×0×1/4×5/6 = 0
False	A lot	False	True	3/4×0×1/4×1/6 = 0
False	A lot	True	False	3/4×0×3/4×1/6 = 0
False	A lot	True	True	3/4×0×3/4×5/6 = 0
True	None	False	False	1/4×1/6×3/4×1/6 = 1/192
True	None	False	True	1/4×1/6×3/4×5/6 = 5/192
True	None	True	False	1/4×1/6×1/4×0 = 0
True	None	True	True	1/4×1/6×1/4×1 = 1/96
True	A little	False	False	1/4×1/3×1/2×1/6 = 1/144
True	A little	False	True	1/4×1/3×1/2×5/6 = 5/144
True	A little	True	False	1/4×1/3×1/2×0 = 0
True	A little	True	True	1/4×1/3×1/2×1 = 1/24

(continued)

Hungry	Eat	Tired	Cry	
True	A lot	False	False	1/4×1/2×1/4×1/6 = 1/192
True	A lot	False	True	1/4×1/2×1/4×5/6 = 5/192
True	A lot	True	False	1/4×1/2×3/4×0 = 0
True	A lot	True	True	1/4×1/2×3/4×1 = 3/32

Exercise 3

$P(Cry) = \Sigma_{Hungry,\ Eat,\ Tired}\ P(Hungry)\ P(Eat \mid Hungry)\ P(Tired \mid Eat)\ P(Cry \mid Hungry, Tired) =$

Cry	
False	15/48+1/48+5/48+1/48+0+0+1/192+0+1/144+0+1/192+0 = 137/288
True	3/48+5/48+1/48+5/48+0+0+5/192+1/96+5/144+1/24+5/192+3/32 = 151/288

Exercise 4

Step 1: Zero out rows where Cry = False

Hungry	Eat	Tired	Cry	
False	None	False	False	0
False	None	False	True	3/48
False	None	True	False	0
False	None	True	True	5/48
False	A little	False	False	0
False	A little	False	True	1/48
False	A little	True	False	0
False	A little	True	True	5/48
False	A lot	False	False	0
False	A lot	False	True	0
False	A lot	True	False	0
False	A lot	True	True	0
True	None	False	False	0
True	None	False	True	5/192
True	None	True	False	0

(continued)

Hungry	Eat	Tired	Cry	
True	None	True	True	1/96
True	A little	False	False	0
True	A little	False	True	5/144
True	A little	True	False	0
True	A little	True	True	1/24
True	A lot	False	False	0
True	A lot	False	True	5/192
True	A lot	True	False	0
True	A lot	True	True	3/32

Step 2: Sum out Hungry and Tired

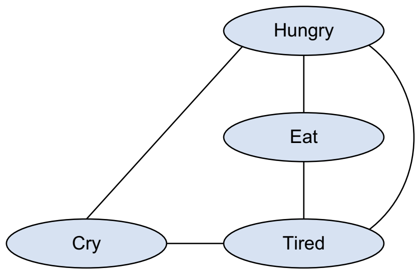
Eat	
None	3/48+5/48+5/192+1/96 = 117/576
A little	1/48+5/48+5/144+1/24 = 116/576
A lot	0+0+5/192+3/32 = 69/576

Step 3: Normalize

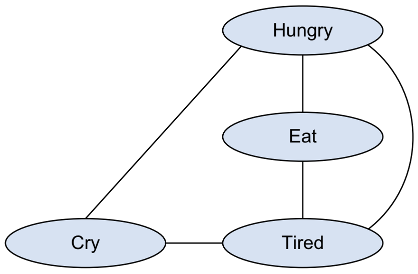
Eat	
None	117/302
A little	116/302
A lot	69/302

Exercise 5

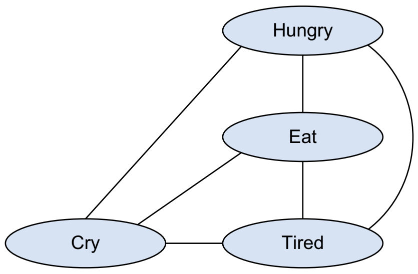
a



b There are two cliques of size 3: Hungry-Eat-Tired and Hungry-Tired-Cry. The induced graph looks like this.



c There is a clique of size 4 containing all the variables. The induced graph looks like this.



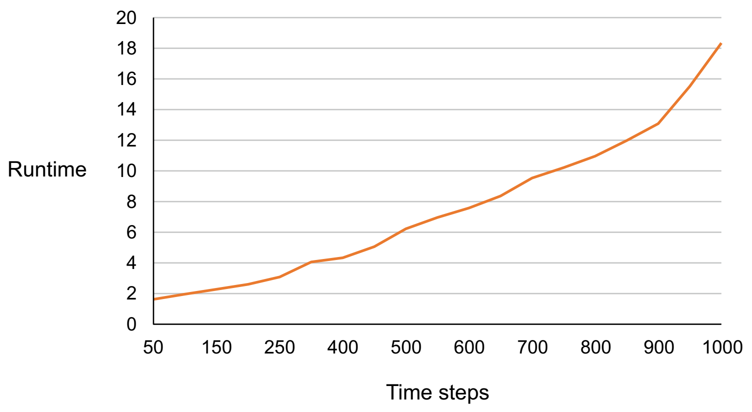
Exercise 7

Here is a program that can be used to measure the performance of variable elimination on this HMM:

```
abstract class State {
  val confident: Element[Boolean]
  def possession: Element[Boolean] =
    If(confident, Flip(0.7), Flip(0.3))
}

class InitialState() extends State {
  val confident = Flip(0.4)
}

class NextState(current: State) extends State {
  val confident =
    If(current.confident, Flip(0.6), Flip(0.3))
}

// produce a state sequence in reverse order of the given length
def stateSequence(n: Int): List[State] = {
  if (n == 0) List(new InitialState())
```

```
    else {
      val last :: rest = stateSequence(n - 1)
      new NextState(last) :: last :: rest
    }
  }

  // unroll the hmm and measure the amount of time to infer the last hidden
     state
  def timing(obsSeq: List[Boolean]): Double = {
    Universe.createNew() // ensures that each experiment is separate
    val stateSeq = stateSequence(obsSeq.length)
    for { i <- 0 until obsSeq.length } {
      stateSeq(i).possession.observe(obsSeq(obsSeq.length - 1 - i))
    }
    val alg = VariableElimination(stateSeq(0).confident)
    val time0 = System.currentTimeMillis()
    alg.start()
    val time1 = System.currentTimeMillis()
    (time1 - time0) / 1000.0 // inference time in seconds
  }

  val steps = 1000
  val obsSeq = List.fill(steps)(scala.util.Random.nextBoolean())
  println(steps + ": " + timing(obsSeq))
```

When I run this program on my computer with steps varying from 50 to 1000 and plot the results, I get the following graph.



We know from the theory that variable elimination is supposed to be linear for an HMM, and indeed we see an approximately linear trend most of the way. However, there's an uptick at around 950 time steps. Why is this? I suspect it has to do with memory issues. My conjecture is that at around 950 time steps, the JVM starts garbage collecting more frequently, and this adds some overhead to the inference.

By the way, at first I tried to run all the experiments for different numbers of time steps in the same program. However, I found that the garbage collection penalty was much more severe as it needed to garbage collect the previous experiment every time it started a new experiment. If possible, when you're running timing experiments, you should run each one separately because you don't know what spillover effect one will have on the next.

Exercise 9

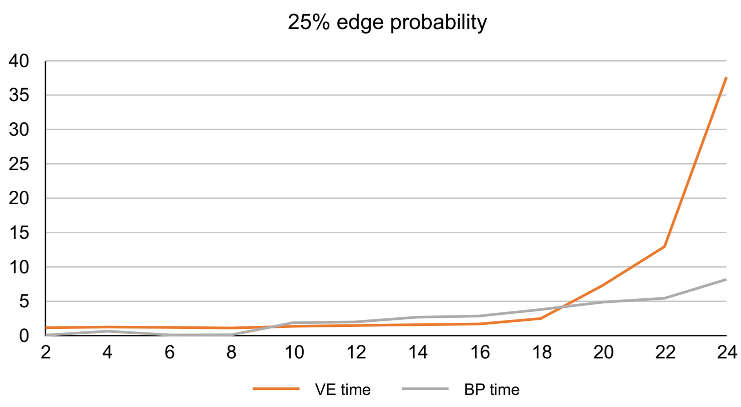
Here's some code that creates a network with a given number of diseases and symptoms and a given probability of each edge, and then runs variable elimination and belief propagation, measuring their runtime and the error of belief propagation. A couple of notes:

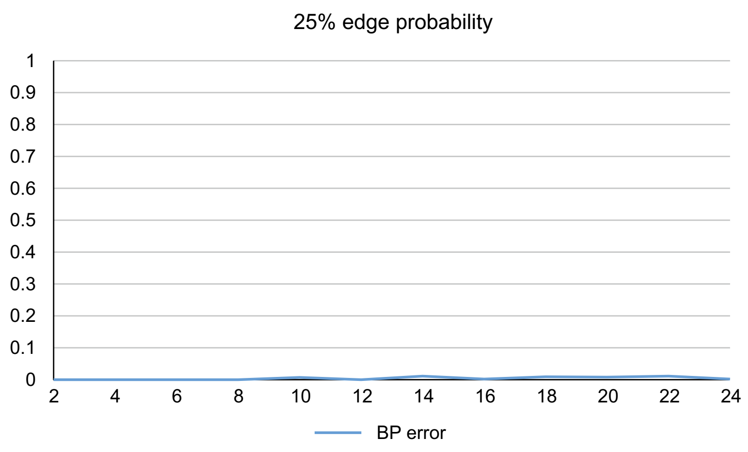
- The symptoms can have any number of disease parents. To represent the dependence of a symptom on its disease parents, I've used the *noisy-or* model. This is a useful model for Boolean variables with many Boolean parents. In the noisy-or model, each disease creates a variable (called `diseaseCauses` in the code), whose effect is to make the symptom definitely true if this node is true. The symptom is then the disjunction of these `diseaseCauses` nodes, implemented using a Figaro container.
- The number of iterations of belief propagation required for good performance depends on the diameter of the network, which is the maximum distance between connected nodes. In this disease-symptom network, the diameter grows as the number of diseases and symptoms grows, so I set the number of iterations of belief propagation to five times the number of diseases plus the number of symptoms, which seemed to work well.

```
val diseases = Array.fill(numDiseases)(Flip(0.1))
def makeSymptom() = {
  val diseaseCauses =
    for {
      i <- 0 until numDiseases
      if scala.util.Random.nextDouble() < pEdge
    } yield {
      diseases(i) && Flip(0.2)
    }
  Container(diseaseCauses:_*).foldLeft(false)(_ || _)
}
val symptoms = Array.fill(numSymptoms)(makeSymptom())

for { i <- 0 until numSymptoms } {
  symptoms(i).observe(true)
}
println(numDiseases + " diseases")
println(numSymptoms + " symptoms")
println(pEdge + " probability of each edge")
val ve = VariableElimination(diseases(0))
val time0 = System.currentTimeMillis()
ve.start()
val time1 = System.currentTimeMillis()
println("Variable elimination time: " + ((time1 - time0) / 1000.0))
```

```
val veAnswer = ve.probability(diseases(0), true)
val bp = BeliefPropagation(5 × (numDiseases + numSymptoms), diseases(0))
val time2 = System.currentTimeMillis()
bp.start()
val time3 = System.currentTimeMillis()
println("Belief propagation time: " + ((time3 - time2) / 1000.0))
val bpAnswer = bp.probability(diseases(0), true)
println("Belief propagation error: " + (math.abs(veAnswer - bpAnswer)))
```

I plotted the runtimes and error as a function of the number of diseases and symptoms (using the same number for each) with edge probability varying from 25% to 100%. Here are the runtimes and error when the edge probability is 25%.

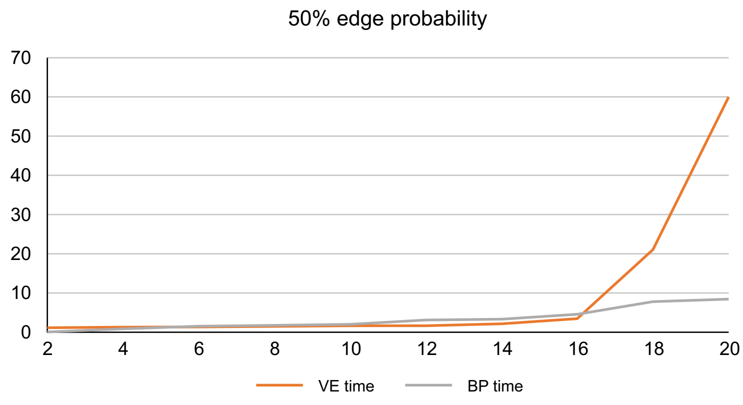


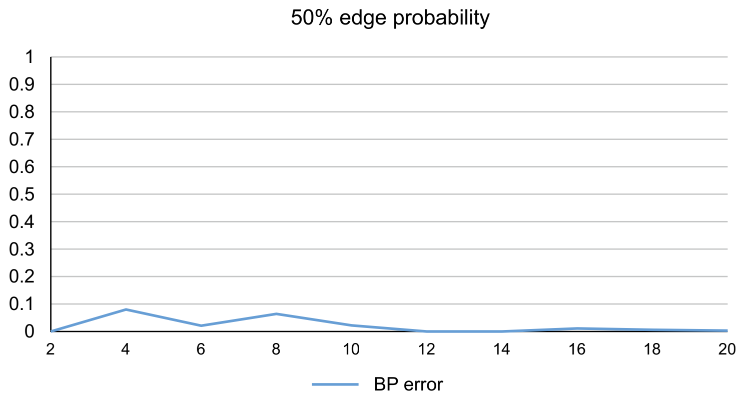


We see that up until about 18 diseases and symptoms, the cost of variable elimination is approximately flat, but then the exponential cost kicks in and rises dramatically.

Belief propagation, in theory, should be quadratic in the number of diseases and symptoms, since the number of iterations and cost of each iteration are proportional to the number of diseases and symptoms. We see that the cost of belief propagation is comparable to variable elimination at first but doesn't increase rapidly after 18 diseases and symptoms. From the second chart, we see that the error of belief propagation is very low throughout, which makes sense because with an edge probability of 25%, there aren't likely to be a lot of loops in the network.

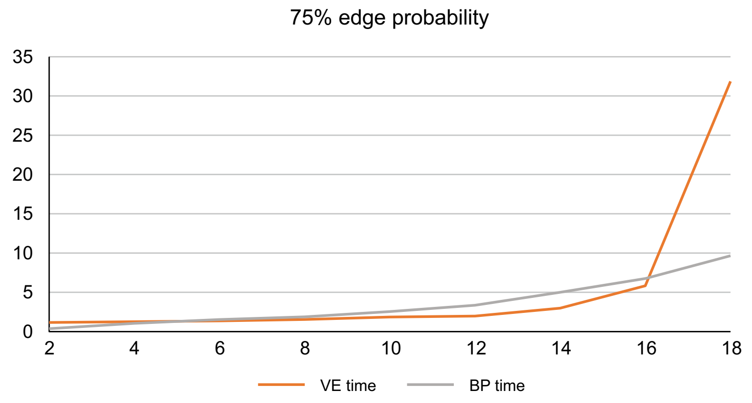
Here are the plots for edge probability 50%.

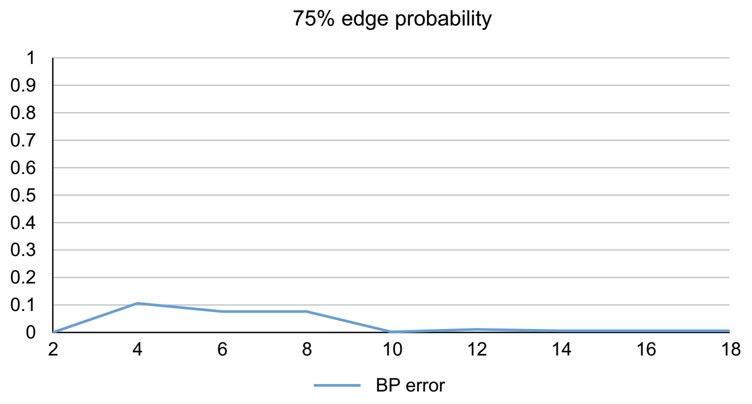




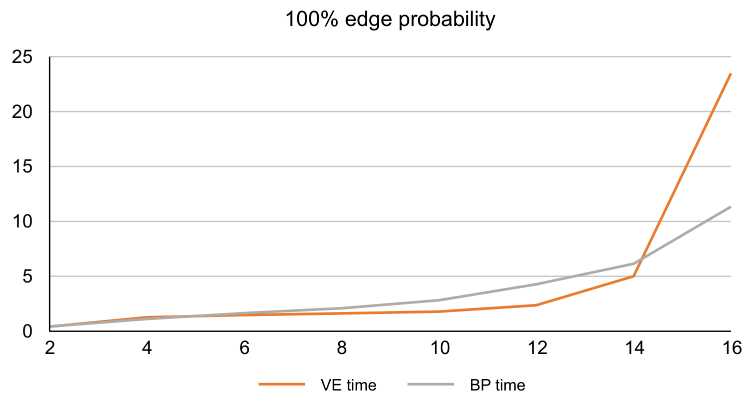
The runtime story is similar, except that the exponential curve kicks in a little earlier, which makes sense since the network is denser. The error plot shows that the error of belief propagation is generally still low but occasionally rises to 10%.

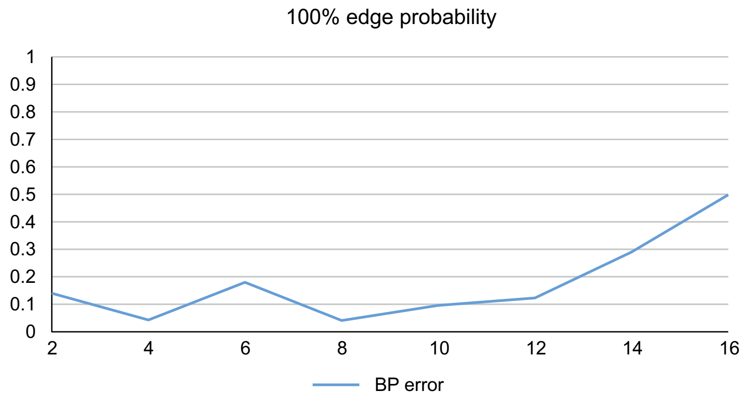
Here are the plots for 75% edge probability.





These results are similar to those for 50% edge probability. Finally, here are the plots for 100% edge probability.





While the runtime results are similar, the error plot is striking as the number of diseases and parents rises to 14 and 16, reaching 50%! This indicates that belief propagation is essentially useless for these networks. The explanation is probably that a fully connected network of diseases and symptoms has an enormous number of loops, rendering belief propagation an inappropriate algorithm.

The bottom line from these experiments is that with an edge probability up to 75%, variable elimination works for small-to-medium-size networks and belief propagation is a viable approximation algorithm for larger networks. For extremely dense networks, variable elimination can handle small networks, but for larger networks, an alternative approximation algorithm should be used.

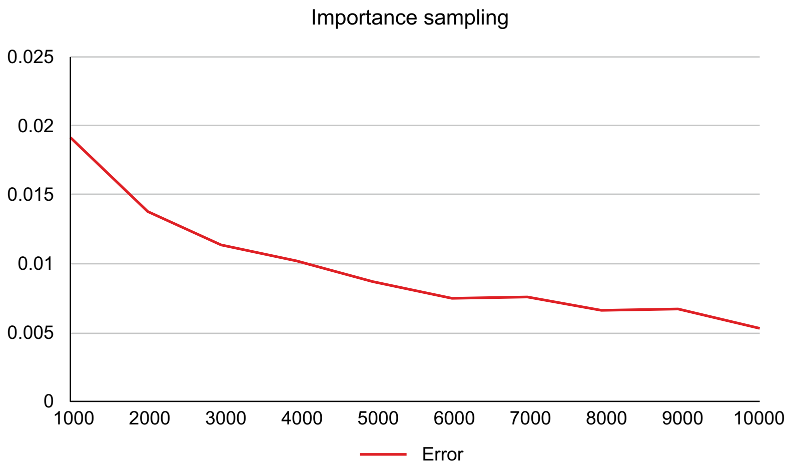
Chapter 11

Exercise 1

Here's a program to run this experiment:

```
val x = Flip(0.8)
val y = Flip(0.6)
val z = If(x === y, Flip(0.9), Flip(0.1))
z.observe(false)

val veAnswer = VariableElimination.probability(y, true)
for { i <- 1000 to 10000 by 1000 } {
  var totalSquaredError = 0.0
  for { j <- 1 to 100 } {
    val imp = Importance(i, y)
    imp.start()
    val impAnswer = imp.probability(y, true)
    val diff = veAnswer - impAnswer
    totalSquaredError += diff × diff
  }
  val rmse = math.sqrt(totalSquaredError / 100)
  println(i + " samples: RMSE = " + rmse)
}
```

When I run this program and plot the RMSE, I get the following graph.



There are some bumps in the graph, showing that even 100 runs is not enough to get a really good estimate of the error. However, there's a clear trend in the graph: the error decreases with more samples, but the rate at which the error decreases slows down. In fact, theory tells us that the error should decrease with the square root of the number of samples, and this graph is consistent with that.

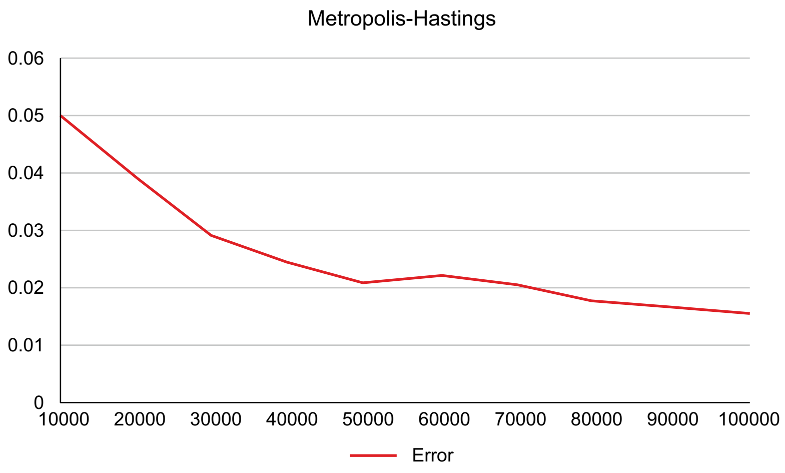
Exercise 2

Here's a program to run this experiment. One note: Metropolis-Hastings makes state so that if you run a new Metropolis-Hastings algorithm in the same universe, it starts with the state of the old one. That means that if you want to run independent experiments, you should run each one in its own universe. This is not something you will ordinarily have to worry about. You'll only be running multiple Metropolis-Hastings algorithms in the same universe if you're running experiments like this.

```
val x = Flip(0.8)
val y = Flip(0.6)
val z = If(x === y, Flip(0.9), Flip(0.1))
z.observe(false)
val veAnswer = VariableElimination.probability(y, true)

for { i <- 10000 to 100000 by 10000 } {
  var totalSquaredError = 0.0
  for { j <- 1 to 100 } {
    Universe.createNew()
    val x = Flip(0.8)
    val y = Flip(0.6)
    val z = If(x === y, Flip(0.9), Flip(0.1))
    z.observe(false)
    val mh = MetropolisHastings(i, ProposalScheme.default, y)
```

```
      mh.start()
      val mhAnswer = mh.probability(y, true)
      val diff = veAnswer - mhAnswer
      totalSquaredError += diff × diff
    }
    val rmse = math.sqrt(totalSquaredError / 100)
    println(i + " samples: RMSE = " + rmse)
  }
```

Here's a plot of the results.



Despite some bumps, we see the same trend as for importance sampling: the error decreases with the square root of the number of samples.

Exercise 3

When I run this experiment, I get an error of about 4×10^{-4} for importance sampling. The reason importance sampling is so accurate despite the low probability of evidence is because it's able to "push the evidence backward" through the program to make sure it samples values consistent with the evidence. In this case, it's able to generate the value false for `z` when `x` is equal to `y` (which is very likely), even though false is a very unlikely value of `z`. Importance sampling then correctly weights the sample with the weight 0.0001.

Exercise 4

In five runs of Metropolis-Hastings on this model, the error ranged from about 0.02 to 0.07. Given that the correct answer is only around 0.09, that's not good at all. The problem is that with the extreme probabilities, it's very hard for Metropolis-Hastings to switch between states. In particular, the state in which `x` is true, `y` is false, the first consequent of the `If` of `z` is true, and the second consequent is false, is the most likely given the evidence. However, from this state, any proposal that proposes to change x or y will be rejected, since that will make x equal to y, in which case the first consequent will apply, which is inconsistent with the evidence. Similarly, changing the value of the second consequent will result in state inconsistent with the evidence. The only possible change that will be accepted from this state is to change the first consequent to false. This will happen only rarely, and what's more, even when it does happen it might be reversed quickly. As a result, Metropolis-Hastings will converge very slowly to the correct distribution.

Exercise 5

Here's the Figaro code to express this proposal scheme:

```
val scheme = DisjointScheme(
    0.1 -> (() => ProposalScheme(z1)),
    0.1 -> (() => ProposalScheme(z2)),
    0.8 -> (() => ProposalScheme(x, y))
)
```

When I run Metropolis-Hastings using this proposal scheme, I get an error around 4×10^{-4}, just like for importance sampling. The reason this works is that now Metropolis-Hastings can switch the values of x and y simultaneously. This means that it can go from the `x = true`, `y = false` configuration to the `x = false`, `y = true` configuration, which it had a very hard time doing under the previous proposal scheme.

Chapter 12

Exercise 1

To query the joint distribution of the printer state and network state given a poor print result and print the joint distribution, you can use the following code:

```
val pair = ^^(printerState, networkState)
printResultSummary.observe('poor)
val ve = VariableElimination(pair)
ve.start()
val dist = ve.distribution(pair).toList
for { (prob, (printerState, networkState)) <- dist } {
  println("Printer " + printerState + ", network " +
           networkState + ": " + prob)
}
```

This prints the following:

```
Printer 'poor, network 'down: 0.0
Printer 'out, network 'intermittent: 0.0
Printer 'good, network 'up: 0.2218637908492195
Printer 'poor, network 'up: 0.5151855275542285
Printer 'out, network 'up: 0.0
Printer 'good, network 'down: 0.0
Printer 'good, network 'intermittent: 0.10186792277702467
Printer 'poor, network 'intermittent: 0.1610827588195274
Printer 'out, network 'down: 0.0
```

First, notice that any state where either the printer is out or the network is down is possible, because in those states, the print result would have been none, not poor. Now, when the printer state is good, the network state is about 2.2 times more likely to be up than intermittent, but when the printer state is poor, the network state is about 3.2 times more likely to be up. Why is this? Well, a poor printer state and an intermittent network state are alternative explanations of the print result summary not being good, so they "explain each other away"—when one explanation holds, the other becomes less likely to hold.

Exercise 2

In the first step, we observe that the printer result is none, run an MPE variable elimination algorithm, and print the most likely states of the possible faults. Here's the code to do this:

```
  printResultSummary.observe('none)
  val alg = MPEVariableElimination()
  alg.start()
  println("Step 1: After observing no print result")
  println("Printer power button on: " +
    alg.mostLikelyValue(printerPowerButtonOn))
  println("Toner level: " + alg.mostLikelyValue(tonerLevel))
  println("Paper flow: " + alg.mostLikelyValue(paperFlow))
  println("Software state: " + alg.mostLikelyValue(softwareState))
  println("Network state: " + alg.mostLikelyValue(networkState))
  println("User command correct: " +
    alg.mostLikelyValue(userCommandCorrect))
```

And here's the result of this code:

```
Step 1: After observing no print result
Printer power button on: true
Toner level: 'high
Paper flow: 'smooth
Software state: 'correct
Network state: 'up
User command correct: false
```

The only fault that is likely to be present is that the user's command is incorrect. Suppose we check the user's command and find that it is indeed correct. We assert the evidence that the user's command is correct, run another MPE variable elimination algorithm, and print the results. Here's what we get:

```
Printer power button on: true
Toner level: 'out
Paper flow: 'smooth
Software state: 'correct
Network state: 'up
User command correct: true
```

Having eliminated the user command, the most likely fault becomes that the toner is out.

Exercise 3

To compute the probability of evidence, we use named evidence. We first need to give the print result summary element a name. This is achieved as follows:

```
val printResultSummary =
  Apply(numPrintedPages, printsQuickly, goodPrintQuality,
        (pages: Symbol, quickly: Boolean, quality: Boolean) =>
        if (pages == 'zero) 'none
        else if (pages == 'some || !quickly || !quality) 'poor
        else 'excellent)("result", Universe.universe)
```

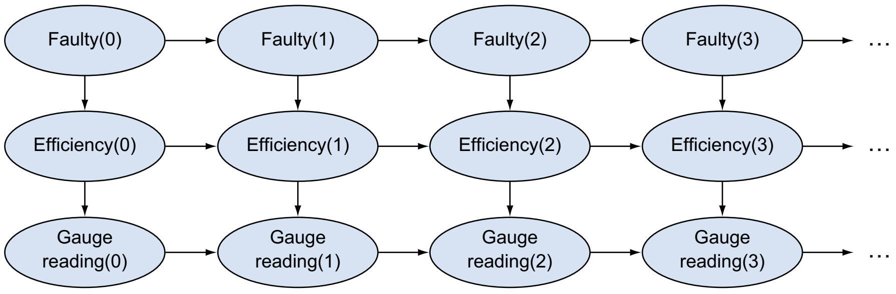
Now we can query for the probability of the evidence that the print result summary is poor as follows:

```
println(ProbEvidenceSampler.computeProbEvidence(10000,
    List(NamedEvidence("result", Observation('poor)))))
```

Chapter 13

Exercise 1

a Since Faulty doesn't change over time, it can be modeled as a single static variable that influences Efficiency at every time point. However, we'll want to query this variable after 100 time steps, so we make it a dynamic variable that can be queried over time. To capture the fact that it doesn't change, we make its conditional probability distribution implement the identity function, i.e., the function that always returns the same value as its input. Here's a DBN to express this model.



b
```
val initial = Universe.createNew()
val faulty = Flip(0.001)("faulty", initial)
val efficiency = Constant('high)("efficiency", initial)
val gauge =
  CPD(efficiency,
    'high -> Select(0.95 -> 'high, 0.05 -> 'low),
    'low -> Select(0.05 -> 'high, 0.95 -> 'low))("gauge", initial)
```

```
def trans(current: Universe): Universe = {
  val next = Universe.createNew()
  val faulty =
    Apply(current.get[Boolean]("faulty"),
          (f: Boolean) => f)("faulty", next)
  val efficiency =
    CPD(current.get[Symbol]("efficiency"), faulty,
        ('low, false) -> Select(0.9 -> 'low, 0.1 -> 'high),
        ('low, true) -> Select(0.9 -> 'low, 0.1 -> 'high),
        ('high, false) -> Select(0.05 -> 'low, 0.95 -> 'high),
        ('high, true) -> Select(0.9 -> 'low, 0.1 -> 'high))(
          "efficiency", next)
  val gauge = CPD(efficiency,
    'high -> Select(0.95 -> 'high, 0.05 -> 'low),
    'low -> Select(0.05 -> 'high, 0.95 -> 'low))("gauge", next)
  next
}
```

c Here's code to generate the observation sequences. I've used importance sampling to generate the sequences with only one sample. This creates a single value at each point in time. To get that value, we can use the first item in the distribution of the gauge reading, which we know has only one positive value.

```
def createObsSeq(carType: Boolean): Array[Symbol] = {
  val obs = Array.fill(100)('low)
  faulty.observe(carType)
  var current = initial
  for { i <- 0 until 100 } {
    current = trans(current)
    val gaugeVar = current.get[Symbol]("gauge")
    val imp = Importance(1, gaugeVar)(current)
    imp.start()
    val obsVal = imp.distribution(gaugeVar)(0)._2
    obs(i) = obsVal
  }
  faulty.unobserve()
  obs
}

val normalObs = createObsSeq(false)
val faultyObs = createObsSeq(true)
```

d Here is code to run the particle filter:

```
val pf1 = ParticleFilter(initial, trans, 100)
pf1.start()
for { i <- 0 until  100 } {
  pf1.advanceTime(List(NamedEvidence("gauge",
                       Observation(normalObs(i)))))
}
println(pf1.currentProbability("faulty", true))

val pf2 = ParticleFilter(initial, trans, 100)
pf2.start()
```

```
for { i <- 0 until  100 } {
  pf2.advanceTime(List(NamedEvidence("gauge",
                       Observation(faultyObs(i)))))
}
println(pf2.currentProbability("faulty", true))
```

When I ran this code ten times, it correctly detected a normal car every time. However, it only detected a faulty car twice in ten attempts. The problem is that with only 100 samples, and with the initial probability of a faulty car being only 1 in 1,000, it's highly likely that the initial set of samples won't contain a single sample where `faulty` is true. The particle filter will never be able to recover from this, no matter how much the evidence points to a faulty car.

e When I ran the particle filter with 10,000 samples, it gets an answer above 0.9 eight times out of ten. This is not perfect but a lot better than before. The reason is that with 10,000 samples, there will be on average ten samples in the initial sample set where `faulty` is true. There's a good chance that some of these will survive and come to dominate the set of samples over time as the evidence points to a faulty car.

Exercise 2

In this case, with 100 samples, the particle filter returns a non-zero probability seven times out of ten. This means that it is usually avoiding particle starvation, where there are no particles where `faulty` is true. However, the returned value is often a long way from 1.0. The reason for this is that in the new model, even if the car was faulty for awhile, it may become normal. So the correct probability of faulty being true at the last time step is lower than it was before.

Exercise 3

a Here's a representation of the model using the `ModelParameters` pattern:

```
val params = ModelParameters()

val xParam = Beta(1, 1)("x", params)
val yGivenXParam = Beta(2, 1)("yGivenX", params)
val yGivenNotXParam = Beta(1, 2)("yGivenNotX", params)
val zGivenYParam = Beta(1, 1)("zGivenY", params)
val zGivenNotYParam = Beta(1, 1)("zGivenNotY", params)

class Model(pc: ParameterCollection) {
  val x = Flip(pc.get("x"))
  val y = If(x, Flip(pc.get("yGivenX")), Flip(pc.get("yGivenNotX")))
  val z = If(y, Flip(pc.get("zGivenY")), Flip(pc.get("zGivenNotY")))
}
```

b
```
 for { i <- 1 to 10 } {
  val xz = scala.util.Random.nextBoolean()
  val model = new Model(params.priorParameters)
```

```
    model.x.observe(xz)
    model.z.observe(xz)
  }
```

c Here's the code to run EM with variable elimination:

```
  val learningAlg = EMWithVE(10, params)
  learningAlg.start()
```

d With variable elimination, I get that the probability of Z being true given that X is true is exactly 0.5. This means that the observation of X has no effect at all on Z, despite the fact that in ten out of ten training examples, the two variables are equal. The explanation for this is that our prior distribution was perfectly symmetric. To account for the correct relationship between X and Z, we would need one value of Y to be much more likely when X is true, and for Z to be much more likely to be true when Y takes that value, and vice versa for the other value of Y. But with a symmetric prior, EM can't learn to associate one value of Y with X being true more than the other. Both values of Y have equal posterior probability in each training example, so the sufficient statistics they generate are equal. As a result, the parameters after each iteration will also be symmetric and no learning will take place.

e Using asymmetric prior distributions solves the problem. In this case, when X is true, Y is more likely to be true, and when X is false, Y is more likely to be false. The training data will reinforce this distinction and also make Z's value correspond to Y. Using EM with variable elimination, I get an answer of about 0.995 for the probability that Z is true given that X is true.

Exercise 4

a Here's a program to perform Bayesian learning on this example using importance sampling. In addition to importance sampling, Metropolis-Hastings also works well on this problem. However, variable elimination and belief propagation don't work well, probably because they don't handle continuous variables well.

```
  val xParam = Beta(1, 1)
  val yGivenXParam = Beta(1, 1)
  val yGivenNotXParam = Beta(1, 1)
  val zGivenYParam = Beta(1, 1)
  val zGivenNotYParam = Beta(1, 1)

  class Model {
    val x = Flip(xParam)
    val y = If(x, Flip(yGivenXParam), Flip(yGivenNotXParam))
    val z = If(y, Flip(zGivenYParam), Flip(zGivenNotYParam))
  }

  for { i <- 1 to 10 } {
    val xz = scala.util.Random.nextBoolean()
    val model = new Model
    model.x.observe(xz)
    model.z.observe(xz)
  }

  val futureModel = new Model
  futureModel.x.observe(true)
  println(Importance.probability(futureModel.z, true))
```

b Bayesian learning is able to learn, even with symmetric Beta(1, 1) priors. The reason for this is that the posterior distribution over the parameters assigns high probability to parameter values that make Z and X likely to be equal. Because the Bayesian learning approach considers the entire distribution and not just a single point, it's able to correctly take into account these high probability regions of the posterior distribution. EM, on the other hand, because it only considers a single point, is unable to find these regions.

Another point of comparison between Bayesian learning and EM is that according to Bayesian learning, the posterior probability of Z being true given that X is true is around 0.75, in contrast to EM where it was close to 1. This is an example of Bayesian learning avoiding overfitting. Just because ten training examples had Z equal to X doesn't mean that's always going to be the case. It would take more training examples like that to fully convince the learner that Z will almost always be equal to X.