1 Programming Overview

1.1 Background

Some students have expressed difficulty translating an algorithm discussed in class into code. It may be worth talking about minimax at a high level. What the general code structure looks like, the fact that it is recursive, and no data structures should be needed with the exception of generating a successor list, etc.

1.2 Minimax

Below is the pseudo code for the algorithm as listed on Wikipedia (https://en.wikipedia.org/wiki/Minimax). Make sure students feel comfortable writing a recursive program. Additionally, it may be worth noting that there is only one for loop, no nested loops. We want them to reduce code complexity as much as possible to write better code.

```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := neg infinity
        for each child of node do
            value := max(value, minimax(child, depth  1, FALSE))
        return value
    else (* minimizing player *)
        value := pos infinity
        for each child of node do
            value := min(value, minimax(child, depth  1, TRUE))
        return value
```

2 Propositional Logic

2.1 Problem Statement

You are the manager of a retail store. Unfortunately, there has been a glitch in the timecard system and all records of who worked when and for how long have been lost for the past 2 months. The timecard company sent an IT specialist out, and they have managed to recover all records with the exception of who worked on the July 4th holiday weekend.

Thankfully, you have a pretty good memory, and you are hoping to put the pieces back together and figure it out yourself. Here's what you remember, the information in parentheses is just to provide relevant backstory:

1. It was July 4th

2. Someone had to have been working (the store was open)
3. If someone was working it must have been John or Alice (they are the only employees)
4. If it was July 4th, John was at the beach (he goes every year)
5. If John was at the beach, he couldn't have been working (beach is far away)

2.2 Try to write out a knowledge base

Ask students to try and take the rules from above and write a logical statement for each one. Below are the corresponding logical statements I came up with:

1. ItWasJuly4th (IWJF)
2. SomeoneWasWorking (SWW)
3. SWW => JohnWasWorking (JWW) OR AliceWasWorking (AWW)
4. IWJF => JohnWasOutOfTown (JWOOT)
5. JWOOT => NOT(JWW)

****Note****: after learning first order logic, there are more convenient ways to describe the following rules. It allows writers to use quantifiers and relations over sentences that contain variables. For example, SomeoneWasWorking can be replaced by "exists x: WasWorking(x)".

2.3 Prove Alice Was Working

Try to get the students to prove that Alice was working given the knowledge base from above (or the one they derived).

Proof:

1. ItWasJuly4th (IWJF)
2. SomeoneWasWorking (SWW)
3. SWW => JohnWasWorking (JWW) OR AliceWasWorking (AWW)
4. IWJF => JohnWasOutOfTown (JWOOT)
5. JWOOT => NOT(JWW)
6. JWW OR AWW (Modus Ponens on 2 and 3)
7. JWOOT (Modus Ponens on 1 and 4)
8. NOT(JWW) (Modus Ponens on 7 and 5)
9. NOT(JWW) => AWW (Equivalent to 6)
10. AWW (Modus Ponens on 8 and 9)

2.4 Proof By Contradiction

Let's assume that Alice was not working, and make sure that it derives a contradiction.

1. ItWasJuly4th (IWJF)
2. SomeoneWasWorking (SWW)
3. SWW => JohnWasWorking (JWW) OR AliceWasWorking (AWW)
4. IWJF => JohnWasOutOfTown (JWOOT)
5. JWOOT => NOT(JWW)
6. NOT(AWW)
7. JWW OR AWW (Modus Ponens on 2 and 3)
8. NOT(AWW) => JWW (Equivalent to 6)
9. JWOOT (Modus Ponens on 1 and 4)
10. JWW (Equivalent to 7)
11. NOT(JWW) (Modus Ponens on 9 and 5)

Having both JWW and NOT(JWW) lets us know this is not possible.

2.5 Writing the Problem in Conjunctive Normal Form (CNF)

You can ask the students to try and write the problem in conjunctive normal form. The book discusses how to do this. Once they have thought a little, you can jump to the following definition (or use one they propose).

- SomeoneWasWorking (SWW)
- NOT(SWW) Or JohnWasWorking (JWW) OR AlicewasWorking (AWW)
- ItWasJuly4th (IWJF)
- NOT(IWJF) OR JohnWasOutOfTown (JWOOT)
- NOT(JWOOT) OR NOT(JWW)

2.6 Prove Alice Was Working Using General Resolution

1. SWW
2. NOT(SWW) OR JWW OR AWW
3. IWJF
4. NOT(IWJF) OR JWOOT
5. NOT(JWOOT) OR NOT(JWW)
6. JWW OR AWW (General resolution on 1 and 2)
7. JWOOT (General resultion on 3 and 4)
8. NOT(JWW) (General resolution on 5 and 7)
9. AWW (General resolution on 6 and 8)

3 Modeling Using Integer Linear Programming

3.1 Note

Ask the students if they are interested in modeling a problem using a linear programming relaxation (one was solved in class so they should be familiar with the concept), and proceed if interest is shown. Otherwise, it isn't absolutely necessary for the course.

3.2 Problem Statement

As discussed in class, linear programming relaxations are linear programs where the integrality constraint of each variable is removed. Programmers can specify, for example that variable x takes some value 0, or 1 with the constraint $x \in \{0, 1\}$.

In class we discussed the satisfiability problem. Maximum satisfiability is like satisfiability, but it seeks to maximize the number of clauses that are true, rather than set all clauses to true. In other words, given a set of clauses in conjunctive normal form, find the maximum number of clauses that can be set to true for all variable assignments. Examples:

$$(x_1) \wedge (\neg(x_1))$$

The maximum satisfiability is 1, as only one clause, not both can ever be true.

$$(x_1) \wedge (\neg(x_1) \vee (x_2))$$

The above statement has maximum satisfiability 2, since setting x_1 to True and x_2 to False results in both clauses being satisfied.

3.3 Linear Programming Relaxation

Note, maximum satisfiability is NP hard. Also, integer and mixed integer linear programs (of which this reduction is) are also NP hard.

3.4 Defining an Objective

In this problem, we want to maximize the number of clauses that evaluate to true. We will assume that we have a helper variable y_c for each clause c which takes on the values $\{0, 1\}$ such that 0 indicates the clause evaluates to false and 1 indicates the clause evaluates to true. Using this, our optimization function then becomes:

$$Max\Big(\sum_c y_c\Big)$$

3.5 Defining Constraints

Let us denote each variable in the problem formula to be x_i. For example the formula

$$(x_1 \vee x_2) \wedge (\neg(x_3))$$

would have three x_i variables, x_1 x_2 and x_3. We want these values to take on True and False. We will denote True to be 1, and False to be 0. Put another way, all x_i must be 0 or 1. This can be stated mathematically by writing $\forall x_i | x_i \in \{0, 1\}$ which reads, "x_i must be 0 or 1 for all x_i". For example, from the formula above, $x_1 \in \{0, 1\}$, $x_2 \in \{0, 1\}$, etc.

Additionally, for each clause, we must calculate y_c for that clause. We want the clause to be set to 1 when any of the expressions being or'd together evaluate to true, and false otherwise. Again, y must take on values 0 or 1. Just ike above, we can say this using the following mathematical notation: $\forall y_c | y_c \in \{0, 1\}$. For example, the first clause $y_1 \in \{0, 1\}$, and the same for y_2, etc.

Lastly, we will denote $X_{c,not}$ to be the set of variables with NOT applied to them in clause c, and $X_{c,reg}$ to be the set of x variables in clause c without NOT applied.

For example, for the clause

$$(x_1 \vee x_3 \vee \neg(x_6))$$

$X_{c,reg} = \{x_1, x_3\}$ and $X_{c,not} = \{x_6\}$. Using this notation we come up with the following constraints:

$$\forall x_i | x_i \in \{0, 1\}$$

$$\forall y_c | y_c \in \{0, 1\}$$

$$y_c \leq \sum_{x_i \in X_{c,reg}} x_i + \sum_{x_i \in X_{c,not}} 1 - x_i$$

1. All variables are either true or false: 0 for false, 1 for true.

2. All clauses are either true or false: 0 for false, 1 for true.

3. A clause can only be true if one of its regular variables was true, or one of its not variables was false. If none of these hold $y_c \leq 0$ mandates that y_c will be false. If at least one of them evaluates to true then we require $y_c \leq 1$, which allows y_c to take on value 1 (True). **Note** that although it appear y_c may get set to 0 even when the clause evaluates to true, this will never happen as the integer linear program is trying to maximize the number of clauses set to true (or the value 1).