15-381: Artificial Intelligence
Assignment 2: Game Theory and Logic

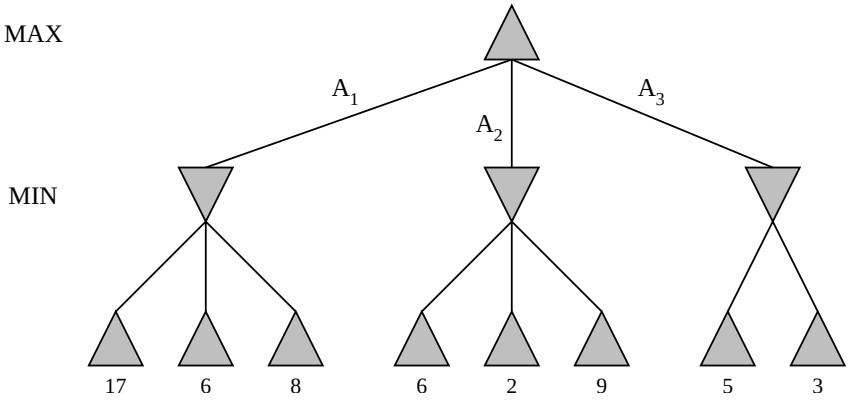
Due: Thursday, September 27, 2001 (before class)

This is the second assignment for 15-381. Similar to the first assignment it contains both written exercises and a programming assignment.

Written Exercises

Exercise 1

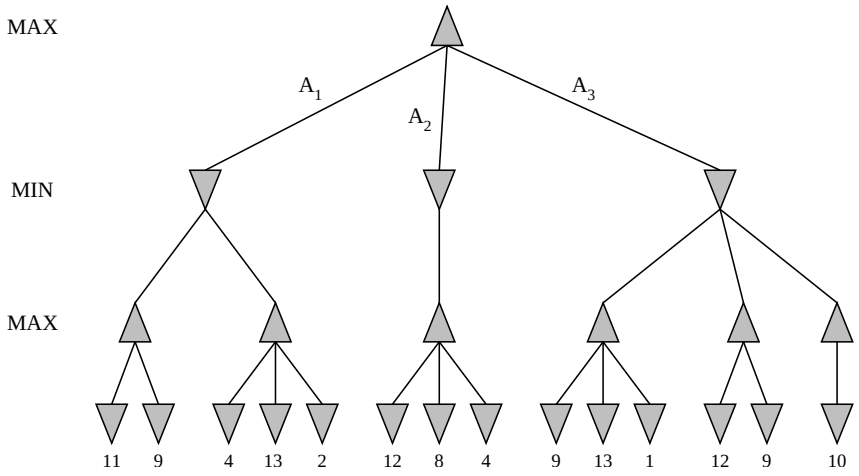
(a) For the game tree given below, show which branches would be pruned by alpha-beta and write down all the node values propagated by the algorithm.



(b) Which action should the MAX player choose?

Exercise 2

(a) For the game tree given below, show which branches would be pruned by alpha-beta and write down all the node values propagated by the algorithm.



(b) Reorder the children of nodes so that the maximum number of nodes are cut from the tree.

Exercise 3

Do exercise 5.8 on page 147 in *Artificial Intelligence: A Modern Approach*.

Programming Assignment

For this programming assignment you will be implementing a theorem prover for a clause logic using the resolution principle[1]. Well-formed sentences in this logic are clauses. As discussed in class, we will be using the disjunctive form instead of the implicative form, since this form is more suitable for automatic manipulation. The syntax of sentences in the clause logic is the following:

$$\begin{aligned} \mathit{Clause} &::= \mathit{Literal} \vee \cdots \vee \mathit{Literal} \\ \mathit{Literal} &::= \neg \mathit{Atom} \mid \mathit{Atom} \\ \mathit{Atom} &::= \mathbf{True} \mid \mathbf{False} \mid P \mid Q \mid \ldots \end{aligned}$$

We will regard two clauses as identical if they have the same literals. For examples, $q \vee \neg p \vee q$, $q \vee \neg p$, and $\neg p \vee q$ are equivalent for our purposes. For this reason we adopt a standardized representation of clauses, with duplicate literals always eliminated.

When modeling real domains, clauses are often written on the form

$$\mathit{Literal} \wedge \cdots \wedge \mathit{Literal} \Rightarrow \mathit{Literal}.$$

[1] You need to be familiar with the components of a logic in order to solve this implementation assignment. If needed, re-read section 6.2, 6.3, and 6.4 in the *Artificial Intelligence: A Modern Approach*.

In this case we need to transform the clauses such that they conform to the syntax of the clause logic. This can always be done using the following simple rules:

1. $p \Rightarrow q$ is equivalent to $\neg p \vee q$
2. $\neg(p \vee q \vee \dots)$ is equivalent to $\neg p \wedge \neg q \wedge \dots$
3. $\neg(p \wedge q \wedge \dots)$ is equivalent to $\neg p \vee \neg q \vee \dots$
4. $(p \wedge q) \wedge \dots$ is equivalent to $p \wedge q \wedge \dots$
5. $(p \vee q) \vee \dots$ is equivalent to $p \vee q \vee \dots$
6. $\neg(\neg p)$ is equivalent to p

The proof theory of the clause logic contains only the resolution rule:

$$\frac{\neg a \vee l_1 \vee \dots \vee l_n \qquad a \vee L_1 \vee \dots \vee L_m}{l_1 \vee \dots l_n \vee L_1 \vee \dots \vee L_m}$$

If there are no literals $l_1, \dots, l_n$ and $L_1, \dots, L_m$ the resolution rule has the form:

$$\frac{\neg a \qquad a}{\textbf{False}}$$

Remember that inference rules are used to generate new valid sentences, given that a set of old sentences are valid. For the clause logic this means that we can use the resolution rule to generate new valid clauses given a set of valid clauses. Consider a simple example where $p \Rightarrow q$, $z \Rightarrow y$, and p are valid clauses. To prove that q is a valid clause, we first need to rewrite the rules to disjunctive form: $\neg p \vee q$, $\neg z \vee y$, and p. Resolution is then applied to the first and the last clause, and we get:

$$\frac{\neg p \vee q \qquad p}{q}$$

If **False** can be deduced by resolution, the original set of clauses is inconsistent. When making proofs by contradiction this is exactly what we want to do. The approach is illustrated by the resolution principle explained below.

The Resolution Principle

To prove that a clause is valid using the resolution method, we attempt to show that the negation of that clause is *unsatisfiable*, meaning it cannot be true under any truth-assignment. This is done using the following algorithm:

1. Negate the clause and add each literal in the resulting conjunction of literals to the set of clauses already known to be valid.
2. Find two clauses for which the resolution rule can be applied. Change the form of the produced clause to the standard form and add it to the set of valid clauses.

3. Repeat 2 until **False** is produced, or until no new clauses can be produced. If no new clauses can be produced, report failure; the original clause is not valid. If **False** is produced, report success; the original clause is valid.

Consider again our example. Assume we now want to prove that $\neg z \vee y$ is valid. First we negate the clause and get $z \wedge \neg y$. Then each literal is added to the set of valid clauses. The resulting set of clauses is:

1. $\neg p \vee q$
2. $\neg z \vee y$
3. p
4. z
5. $\neg y$

Resolution on 2 and 5 gives:

1. $\neg p \vee q$
2. $\neg z \vee y$
3. p
4. z
5. $\neg y$
6. $\neg z$

Finally, we apply the resolution rule on 4 and 6, which produces **False**. Thus, the original clause $\neg z \vee y$ is valid.

Part I: The Program

Your program should read a text file containing the initial set of valid clauses, and a clause for each literal in the negated clause that we want to test validity of. Each line in the file defines a single clause. The literal of each clause are separated by a blank space, and "$\sim$" is used to represent negation.

Your program should implement the resolution algorithm as specified in the previous section. The output is "Invalid clause" if the clause can not be shown to be valid, or the list of clauses in the proof tree for deducing **False**. In either case you should also output the size of the final set of valid clauses.

Let us consider a correct solution for testing the validity of $\neg z \vee y$ in our earlier example. The input file would be:

```
~p q
~z y
p
z
~y
```

A possible output could be:

```
1. ~p q  {}
2. ~z y  {}
3. p     {}
4. z     {}
5. ~y    {}
6. ~z    {2,5}
7. False {4,6}
Size of final clause set: 7
```

Note how the program keeps track of the parents of new clauses. This is used for extracting the clauses in the proof tree.

Part II: Loki in Pittsburgh?

It is now time to put your resolution prover to work! In Norse mythology there are gods and giants. One person, Loki, is both a giant and a god, so if you meet someone who is both of these you know you have met Loki. A brave person who is violent and not admired is a giant. If a person is violent but not brave, or neither brave nor admired, then that person is also giant. A person who is both wise and handsome must surely be a god. A one-eyed person is wise because he has given one of his eyes to drink from the spring of wisdom. A person who still has both eyes, but who is not violent, is also wise. Only handsome people can get married. If you are not married, but admired, then you are for certain handsome. Formally, these rules can be described by the following clauses:

$$
\begin{aligned}
\mathit{God} \wedge \mathit{Giant} &\Rightarrow \mathit{Loki} \\
\mathit{Brave} \wedge \mathit{Violent} \wedge \neg \mathit{Admired} &\Rightarrow \mathit{Giant} \\
\neg \mathit{Brave} \wedge \mathit{Violent} &\Rightarrow \mathit{Giant} \\
\neg \mathit{Brave} \wedge \neg \mathit{Admired} &\Rightarrow \mathit{Giant} \\
\mathit{Wise} \wedge \mathit{Handsome} &\Rightarrow \mathit{God} \\
\mathit{OneEyed} &\Rightarrow \mathit{Wise} \\
\neg \mathit{OneEyed} \wedge \neg \mathit{Violent} &\Rightarrow \mathit{Wise} \\
\mathit{Married} &\Rightarrow \mathit{Handsome} \\
\neg \mathit{Married} \wedge \mathit{Admired} &\Rightarrow \mathit{Handsome}
\end{aligned}
$$

Both gods and giants (and Loki of course too, since he is both) often wander around among us mortals. On such occasions they wear a disguise so as not to draw any attention. One day you meet a violent one-eyed person in a bar in Pittsburgh, who is being admired by a group of people around him. It is quite obvious to your observant eye that this person is neither married nor particularly

brave. Formally, the following literals hold for this person:

$$\begin{aligned} &\mathit{Violent} \\ &\mathit{OneEyed} \\ &\mathit{Admired} \\ \neg&\mathit{Married} \\ \neg&\mathit{Brave} \end{aligned}$$

These literals define the current state.

1. Rewrite the safety rules from their implicative form to the disjunctive form used by your resolution prover. The initial set of valid clauses is the union of the rule clauses and the clauses defining the current state. Write the clauses in a file called `mythology.txt`.

2. Use your resolution prover to test whether the person you see in the bar is a giant.

   (a) Save the input in a file called `giant.in`.

   (b) Save the output from your prover in a file called `giant.out`.

3. Now test if the person is Loki.

   (a) Save the input in a file called `loki.in`.

   (b) Save the output from your prover in a file called `loki.out`.

4. Consider the following subset of the rules:

$$\begin{aligned} \mathit{Wise} \wedge \mathit{Handsome} &\Rightarrow \mathit{God} \\ \mathit{OneEyed} &\Rightarrow \mathit{Wise} \\ \neg\mathit{OneEyed} \wedge \neg\mathit{Violent} &\Rightarrow \mathit{Wise} \\ \mathit{Married} &\Rightarrow \mathit{Handsome} \\ \neg\mathit{Married} \wedge \mathit{Admired} &\Rightarrow \mathit{Handsome} \end{aligned}$$

   Test the validity of $\neg\mathit{God}$ for a one-eyed person who is admired but not married.

   (a) Save the input in a file called `not_god.in`.

   (b) Save the output from your prover in a file called `not_god.out`.

Requirements

We strongly encourage you to write your program in C++, using the standard container classes to represent generic data structures such as sets, vectors, and lists. You are allowed to use C or Java as well. To help you along the way we have written a data structure for clauses and a function for reading an input file. We also provide a program for showing how to manipulate sets in C++. These files can be found in:

/afs/andrew.cmu.edu/course/15/381/homework/Project2/samples/

You must turn in documented source code, as well as an executable compiled under Linux. Your executable should be named `resolution` and take the name of the input file as a command-line argument. All output should be written to standard output. Also hand in a copy of the files mentioned in the "Loki in Pittsburgh?" section. If you are using Java, create an executable script that starts your program. This could look as follows:

```
#!/bin/sh
java Resolution $*
```

Extra Credit

Any implementation conforming to the input format and outputting the correct solutions will get full credit. In Addition, you will receive extra credit for:

1. Lowering the complexity of the basic algorithm. (For example by providing a fast approach for checking whether a new clause already exists in the set of valid clauses, or to find clauses to do resolution on.)

2. Coming up with heuristics for choosing which clauses to do resolution on in order to generate a proof in fewer steps.

Submission Instructions

After completing the assignment, place one copy of your written answers, source code, executable and input/output files in:

/afs/andrew.cmu.edu/course/15/381/handin/*your_id*/p2/

These directories will be created a few days before the due date and will be closed when class starts on Thursday, September 27, 2001. If you are submitting late, let us know and we will keep the folder open. E-mail Håkan (lorens@cs.cmu.edu) when you do submit it.