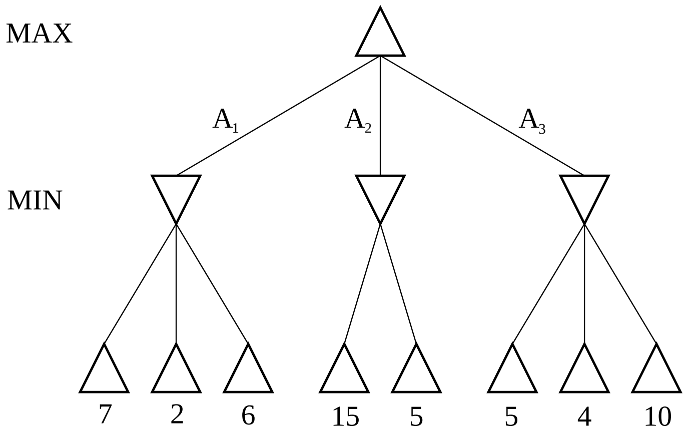
# 15-381 Problem Set 2: Game Theory and Logic

Out: Friday, September 15, 2000
Due: Thursday, September 28, 2000

This is the second problem set for 15-381. Similar to the first problem set it contains both written exercises and a programming assignment. New groups have been generated for this assignment. Please check the class web page to find your new partner. Send email to Rune (runej@cs) if you need to be reassigned to a new group, or if you for some reason were not assigned a partner.

## Written Exercises

### Exercise 1




Questions

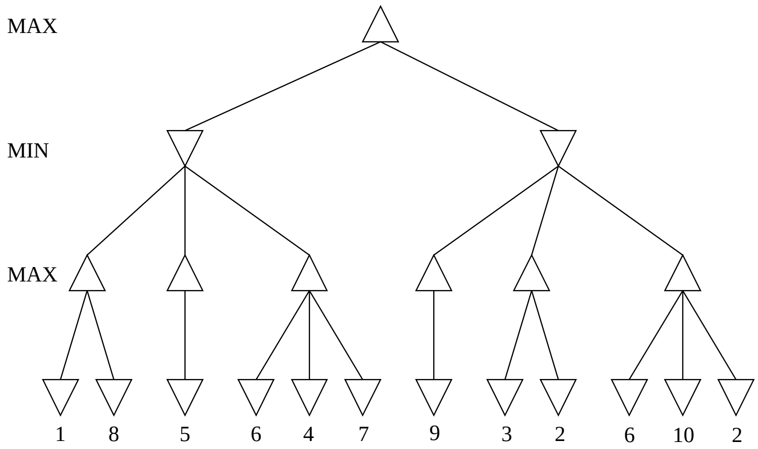
**(a)** Show which branches alpha-beta prunes from the game tree and write the node values propagated by the algorithm.

**(b)** Which action, $A_1$, $A_2$ or $A_3$, should the player choose ?

### Exercise 2

Questions

**(a)** Show which branches alpha-beta prunes from the game tree and write the node values propagated by the algorithm.




**(b)** Rearrange children nodes such that the maximum number of nodes are cut from the tree.

### Exercise 3

Exercise 5.6, pg. 147 in *Artificial Intelligence: A Modern Approach*.

### Exercise 4

Exercise 5.8, pg. 147 in *Artificial Intelligence: A Modern Approach*.

## Resolution Theorem Prover Implementation

In 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. Instead of using the implicative form, as described in class, we will be using the disjunctive form, since this form is more suitable for automatic manipulation. The syntax of sentences in the clause logic is thus:

$$\begin{aligned} \mathit{Clause} &\rightarrow \mathit{Literal} \vee \cdots \vee \mathit{Literal} \\ \mathit{Literal} &\rightarrow \neg \mathit{Atom} \mid \mathit{Atom} \\ \mathit{Atom} &\rightarrow \textbf{True} \mid \textbf{False} \mid P \mid Q \mid \cdots \end{aligned}$$

We will regard two clauses as identical if they have the same literals. For example, $q \vee \neg p \vee q$, $q \vee \neg p$, and $\neg p \vee q$ are equivalent for our purposes. For

[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 *Artificial Intelligence: A Modern Approach.*

this reason we adopt a standardized representation of clauses, with duplicated literals always eliminated.
When modelling real domains, clauses are often written on the form:

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

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. $(\neg(p \vee q))$ is equivalent to $(\neg p \wedge \neg q)$
3. $(\neg(p \wedge q))$ is equivalent to $(\neg p \vee \neg q)$
4. $((p \wedge q) \wedge \cdots)$ is equivalent to $(p \wedge q \wedge \cdots)$
5. $((p \vee q) \vee \cdots)$ is equivalent to $(p \vee q \vee \cdots)$
6. $(\neg(\neg p))$ is equivalent to $p$

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

$$\frac{\begin{array}{c}\neg a \vee l_1 \vee \cdots \vee l_n, \\ a \vee L_1 \vee \cdots \vee L_m\end{array}}{l_1 \vee \cdots \vee l_n \vee L_1 \vee \cdots \vee L_m}$$

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

$$\frac{\neg a,\ a}{\mathbf{False}}$$

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 last clause, and we get:

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

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 the 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 (see 4. and 5.). 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. 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 literals of each clause are separated by a blank space and "~" is used to represent negation.

Your program should implement the resolution algorithm as explained in the previous section. The output is either "Failure" 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 return the size of the final set of valid clauses.

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

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

A possible final set of valid sentences could be:

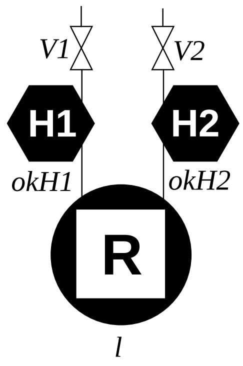
```
1.  ~p q   {}
2.  ~z y   {}
3.   p     {}
4.   z     {}
5.  ~y     {}
6.  ~z     {2,5}
7.   False {4,6}
```

Note how the program keeps track of the parents of new clauses. This is used for extracting the clauses in the proof tree. The solution you return consists of these clauses and the size of the final set of valid clauses:

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

## Part II: Power Plant Diagnosis

In the last part of this problem set you will be using your resolution prover to verify the safety requirements of a reactor unit in a nuclear power plant. The reactor unit is shown in the below figure and consists of a reactor $R$, two heat exchangers $H1$ and $H2$, two steam valves $V1$ and $V2$, and a control stick $l$ for changing the level of energy production. The state of the reactor unit is given by 5 propositional variables $l$, *okH1*, *okH2*, *V1* and *V2*. If $l$ has the




value **True** the production level is 2 energy units. Otherwise, the production level is 1 energy unit. At least one working heat exchanger is necessary for each energy unit produced to keep the reactor from overheating. Unfortunately a heat exchanger $i$ can start leaking reactive water from the internal cooling system to the surroundings. $okHi$ is **False** if heat exchanger $Hi$ is leaking. Otherwise, $okHi$ is **True**. When a heat exchanger $i$ is leaking, it must be shut off by closing its valve $Vi$. The state variable $Vi$ indicates, whether the valve $Vi$ is closed (**False**) or open (**True**). Formally the safety requirements are described by the following clauses:

$$
\begin{array}{rcl}
NoLeak \wedge LowTemp & \Rightarrow & ReactorUnitSafe \\
NoLeakH1 \wedge NoLeakH2 & \Rightarrow & NoLeak \\
okH1 & \Rightarrow & NoLeakH1 \\
\neg okH1 \wedge \neg V1 & \Rightarrow & NoLeakH1 \\
okH2 & \Rightarrow & NoLeakH2 \\
\neg okH2 \wedge \neg V2 & \Rightarrow & NoLeakH2 \\
l \wedge V1 \wedge V2 & \Rightarrow & LowTemp \\
\neg l \wedge V1 & \Rightarrow & LowTemp \\
\neg l \wedge V2 & \Rightarrow & LowTemp
\end{array}
$$

Assume that the current state of the reactor unit is given by the clauses:

$$
\begin{array}{r}
\neg l \\
\neg okH2 \\
okH1 \\
V1 \\
\neg V2
\end{array}
$$

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 `facts.txt`.

2. Use your resolution prover to test whether *LowTemp* is a valid clause:

   (a) Save the input in a file called `task1.in`.

   (b) Save the result of you prover in a file called `task1.out`.

3. Now test the validity of *ReactorUnitSafe* in a similar way:

   (a) Save the input in a file called `task2.in`.

   (b) Save the result of you prover in a file called `task2.out`.

4. Consider a simpler set of safety rules:

$$
\begin{aligned}
\mathit{NoLeakH1} \wedge \mathit{NoLeakH2} &\Rightarrow \mathit{NoLeak} \\
\mathit{okH1} &\Rightarrow \mathit{NoLeakH1} \\
\neg \mathit{okH1} \wedge \neg \mathit{V1} &\Rightarrow \mathit{NoLeakH1} \\
\mathit{okH2} &\Rightarrow \mathit{NoLeakH2} \\
\neg \mathit{okH2} \wedge \neg \mathit{V2} &\Rightarrow \mathit{NoLeakH2}
\end{aligned}
$$

   and a reduced current state description:

$$
\begin{array}{r}
\neg \mathit{okH2} \\
\mathit{okH1} \\
\neg \mathit{V2}
\end{array}
$$

   Test the validity of $\neg \mathit{NoLeak}$

   (a) Save the input in a file called `task3.in`.

   (b) Save the result of you prover in a file called `task3.out`.

## Requirements

The program must be written in C/C++. We strongly encourage you to use the C++ Standard Template Library (STL) to represent generic data structures such as sets, vectors and lists. 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 showing how to manipulate sets using STL. These files can be found in:

`/afs/cs/academic/class/15381-f00/public/hw2-testfiles/`

You and your partner should hand in a single set of solutions to the written exercises and a single copy of your resolution prover. You must turn in documented source code, as well as an executable compiled under Linux. Your executable should take the name of the input file as a command-line argument. Also hand in a copy of the files mentioned in the power plant diagnosis exercises.

### Extra Credit

Any implementation conforming to the input format and returning 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 using a more efficient representation of atoms other than strings or by providing a fast approach for checking whether a new clause already exists in the set of valid clauses.)

2. Coming up with heuristics for choosing which clauses to do resolution on in order to generate a proof in a small number of iterations.

## Submission Instructions

After completing the assignment, place one copy of your written answers, source code, executable and input/output files in:
`/afs/andrew/scs/cs/15-381/handin/`*userid1_userid2*

Submission directories for each group are available by next Tuesday (9/19), and will be closed at 11:59pm on September 28th.

Please contact either Laurie (hyaku@cs) or Rune (runej@cs) if you have any questions or encounter difficulties.