Minesweeper

Ryan McIntyre

December 2016

1 Background

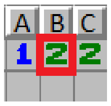
Minesweeper is a game in which the player attempts to clear a grid containing a known number of "mined" squares. Initially, the player knows nothing about the locations of the mines, and must infer their locations through the information provided by the game. Each unmined square contains a number which reflects the number of mines in the eight squares adjacent, but the player cannot see the number in a given square until they have cleared that square. The goal is to clear all of the unmined squares without setting off any mines. The player does so by using the numbers in cleared locations to infer the locations of mines (and, optionally, 'flagging' known mine locations for future reference), and then using inferred mine locations and the same numbers to infer which locations do not contain mines. Newly cleared locations contain numbers of their own, granting the player more information about mine locations... etc.

2 Goals

The goal of this project was to implement first order logic in the form of a Minesweeper player in Python. Initially, the player was intended to play the current Windows version of the game, but since Minesweeper is no longer included in Windows 10, my implementation plays my text-based version of Minesweeper in the iPython console. In terms of game mechanics, the only difference between my Minesweeper and the Windows' is as follows: mine sets up the game board (distributes mines, etc) and then chooses a safe starting location for the player, whereas Windows' lets the player choose a location, and then distributes mines in such a way that the player's chosen location is safe. Regardless of the medium through which the game is played, the player should play optimally on arbitrarily large grids, and only guess when forced to do so.

3 Algorithms and Methods

Given a partially cleared board, the player must first form a knowledge base that reflects the information provided. This knowledge base takes the form of a set of clauses, where each clause is a disjunction (so the conjunction of the clauses is the conjunctive normal form of the information on the board. This could be done, slowly and painfully, by enumerating every possible setup on the board and then converting the result to conjunctive normal form, but this particular application has a convenient shortcut, which I will demonstrate with an example:



Consider the "2" outlined in red. This cleared square has only 3 adjacent uncleared squares, labeled **A,B,C**. This 2 tells us that exactly two of its three adjacent squares contain mines, and as such that exactly one is safe. The goal is to convert the information provided into a set of disjunctions (or clauses). For simplicity of notation, we'll structure our statements in the following way:

A: location **A** contains a mine

¬**A**: location **A** does not contain a mine

First, we'll aim to generate clauses which reflect the possibilities regarding which squares **do** contain a mine. Since exactly one square is safe, we can simply choose any two and be guaranteed to encounter at least one mine. ${}_3C_2 = 3$, so this grants three clauses:

$$\mathbf{A} \vee \mathbf{B} \tag{1}$$

$$\mathbf{A} \vee \mathbf{C} \tag{2}$$

$$\mathbf{B} \vee \mathbf{C} \tag{3}$$

Next, clauses which reflect possibilities regarding which squares **do not** contain a mine. There are two mines, so choosing three squares guarantees that at least one is safe. ${}_3C_3 = 1$, so this gives us one additional clause:

$$\neg\mathbf{A} \vee \neg\mathbf{B} \vee \neg\mathbf{C} \tag{4}$$

The player goes through this process for every cleared square with adjacent uncleared squares to gather its knowledge base. Of course, many of the cleared squares grant immediate information pertaining to the presence or lack thereof of mines. If the number in a square is equal to that squares number of adjacent uncleared squares, then obviously all of said squares contain mines. Similarly, if the number in a square is equal to the number of adjacent flagged squares (with the assumption that all prior flags are correct), obviously none of the remaining adjacent squares contain mines. Information of this type need not be stored in the knowledge base. The player simply clears or flags the remaining squares in question accordingly, and no inference is necessary.

Of course, our domain of interest is when none of these "automatic" moves remain, and the player must infer additional knowledge from its knowledge base. It does so in two ways: Resolution and Forward Chaining. The core of each of these methods is the Resolve algorithm, which allows generation of new clauses from existing ones. The algorithm is quite simple. It takes as input two clauses from the knowledge base, C_1 and C_2. For every statement $\mathbf{A}$ in C_1, it checks if the negation $\neg\mathbf{A}$ is in C_2. If it is, then a new clause can be formed from the disjunction everything in C_1 and C_2 other than $\mathbf{A}$ and $\neg\mathbf{A}$. Consider the clauses below:

$$\mathbf{A} \vee \mathbf{C} \tag{5}$$

$$\neg\mathbf{A} \vee \neg\mathbf{B} \tag{6}$$

From these two clauses, Resolve outputs the third clause:

$$\neg\mathbf{B} \vee \mathbf{C} \tag{7}$$

Forward Chaining runs the resolve algorithm on every pair of clauses in the knowledge base in an attempt to generate new clauses. If it finds new clauses, it runs on every pair in the newly increased knowledge base until it can no longer generate any new information from its current information. Its goal is to find new clauses containing exactly one statement. When it finds clauses of this type, the conclusion is that said statement is true; the disjunction of a single statement has the same truth value as that statement.

Mechanically speaking, Resolution works in a very similar manner. There are two key differences, however. First, Resolution starts by identifying a square of interest, ($\mathbf{A}$) which is uncleared but is adjacent to cleared squares (so it is in the knowledge base). It assumes that $\mathbf{A}$ does not contain a mine by adding the statement $\neg\mathbf{A}$ and then starts generating new statements in the same manner that Forward Chaining does, but with a different goal: finding a contradiction. If it finds two contradictory clauses of the form $\mathbf{B}$ and $\neg\mathbf{B}$ and attempts to Resolve them, it gets an empty clause. The assumption in this case is that the knowledge base was sound prior to the assumption $\neg\mathbf{A}$, and as such said assumption must be false: $\mathbf{A}$ contains a mine. If this fails to grant a contradiction, it will then assume make the opposite assumption in an attempt to prove through contradiction that $\neg\mathbf{A}$ is true (ie $\mathbf{A}$ does not contain a mine). If this too fails to lead to a contradiction, it will move on to the next location of interest.

Of course, these are the idealized versions of Resolution and Forward Chaining, and their implementations have a few extra limits imposed on them. Both begin to run slowly when the knowledge base reaches approximately 300 clauses, so they both stop once they exceed this value. This is fine for small games, but our aim is to work on arbitrarily large grids. My solution was to localise the knowledge base that each uses. Each will choose a location of interest (for Resolution, the locations of interest described above; for Forward Chaining, any square whose grid coordinates are both multiples of 5). They will then generate a knowledge base from the cleared squares in an area around the location of interest, and expand that area only when they exhaust the ability to generate new clauses but have not yet reached their clause limit. This is not ideal, as it prohibits longer chains of inference. But in practice, it tends to work very well given the local nature of the information provided in Minesweeper.

It may be of interest that Resolution never finds any new information if it is run after Forward Chaining (though it works perfectly well, and does generate correct information). The obvious suspicion is that Forward Chaining will find any information that Resolution would have found. In light of this, I have given Resolution a larger clause limit than Forward Chaining; Forward Chaining is used for a quick scan, looking for easy inferences, while Resolution is used as a more detailed last resort.

I'll end this section with the pseudo-code for the three algorithms mentioned in above. It should be noted that this particular implementation stores clauses as sets, and that these sets are treated as disjunctions of their elements.

Algorithm 1 Resolve

```
Input: Pair of sets $C_1, C_2$
Output: Set of new clauses $N$
1: $N \leftarrow \emptyset$
2: for Statement $\mathbf{A}$ in $C_1$ do
3:     if $\neg$ $\mathbf{A}$ in $C_2$ then
4:         $n \leftarrow (C_1 \cup C_2) - \{\mathbf{A}, \neg\mathbf{A}\}$
5:         $N \leftarrow N \cup \{n\}$
6:     end if
7: end for
8: return $N$
```

Algorithm 2 Forward Chaining (as used in this particular game)

```
Input: Knowledge Base $KB$
1: $C \leftarrow KB$
2: $N \leftarrow \emptyset$
3: loop do
4:     for Pair of clauses $C_1, C_2$ in $C$ do
5:         $R \leftarrow$ Resolve$(C_1, C_2)$
6:         $R \leftarrow R - C$
7:         for clause $C_n$ in $R$ do
8:             if $C_n$ contains only one statement then
9:                 Add the corresponding move to the player's move queue
10:            end if
11:        end for
12:        $N \leftarrow N \cup R$
13:    end for
14:    if $N \subseteq C$ then
15:        break
16:    end if
17:    $C \leftarrow C \cup N$
18: end loop
```

Algorithm 3 Resolution

```
Input: Knowledge Base $KB$, statement $\alpha$
Output: True if $KB$ entails $\alpha$; False otherwise
 1: $C \leftarrow KB \cup \{\{\neg\alpha\}\}$ (the inner set is the "disjunction" of the one statement)
 2: $N \leftarrow \emptyset$
 3: loop do
 4:     for Pair of clauses $C_1, C_2$ in $C$ do
 5:         $R \leftarrow$ Resolve($C_1, C_2$)
 6:         if $R$ contains $\emptyset$ then
 7:             return True
 8:         end if
 9:         $N \leftarrow N \cup R$
10:     end for
11:     if $N \subseteq C$ then
12:         return False
13:     end if
14:     $C \leftarrow C \cup N$
15: end loop
```

4 Results

The player works reasonably well, but it does have two drawbacks. First, though it does occasionally fail to draw conclusions that are the result of longer chains of inference (even within a local knowledge base). I believe that this is due to the limits placed on the number of clauses for Resolution and Forward Chaining. Second, these forms of inference are still somewhat slow when used on larger boards, even with localised knowledge bases (as the board size effects the number of these local knowledge bases which must be formed and evaluated). Either of these problems could be eliminated, but doing so would make the other worse; it's just a matter of finding a balance between speed and performance.