Winning World War II with constraint programming

A practical on constraint modeling

May 15, 2018

Installation

You can do the practical using either Choco (Java) or Numberjack (Python). I recommend to use Choco because you are less likely to run into bugs and other problems.

Choco (recommended)

Download Choco from there
`https://github.com/chocoteam/choco-solver/releases/tag/4.0.6`

Makefile method (I can help)

- copy `choco-solver-4.0.6-with-dependencies.jar` into practical/Choco
- use `make` to build and `./run Warmup` to run Warmup.java

IDE method (you're on your own)

- Install it following the instructions on Choco's website (`http://www.choco-solver.org/`) in your preferred IDE
- Then you add the files from the practical to a Choco project

Resources: Choco's user guide and Javadoc.

Numberjack (nicer, but riskier)

Makefile method (I can help)

- run `pip install Numberjack`
  (and pray)

Resources: Numberjack/userguide.pdf.

Warmup (`Warmup.java`)

Let's start a with simple *transposition* code. A transposition code uses a static transposition table: a permutation of the alphabet. Here we pretend to be spies and we want to use constraint programming to encrypt and decrypt messages.

Exercice 1 `transposeEncodeChar`: Write a model with two variables, one standing for a text letter and another standing for the ciphered message by the given transposition table.(search is already setup to find all feasible combinations)

Hint: you might want to use a global constraint very briefly mentioned in yesterday's lecture

Exercice 2 `transposeEncodeText`: Write a model with one variable per letter in the ciphertext. The solution will correspond to the result of encrypting the plaintext "IAMAREALSPYNOW" with the same transposition table.

Exercice 3 `transposeDecodeText`: Write a model with one variable per letter in the plaintext. The solution will correspond to the message that was encrypted with the same transposition table and yielding the ciphertext "MYXXRPQWGYXFRUYLRLMYOECYPE"

Breaking commercial Enigma (`CommercialEnigmaBreaker.java`)

We haven't broken the transposition code, but this is relatively easy using statistical methods because it conserves all the patterns of the language, which we leave for another practical. We are going to break Enigma's code, however!

The commercial Enigma machine is significantly harder to break. It is based on *rotors* which are rotating transposition tables. An Enigma machine has a *keyboard*, three *rotors*, a *reflector* and a *display*. When a key is pressed ('A' in the example in figure 1), the characters traverses the three rotors once *forward*, then it is transposed to another character by the reflector (a transposition table that is symmetric and irreflexive), then it traverses the three same rotors *backward* to finally appear on the display.

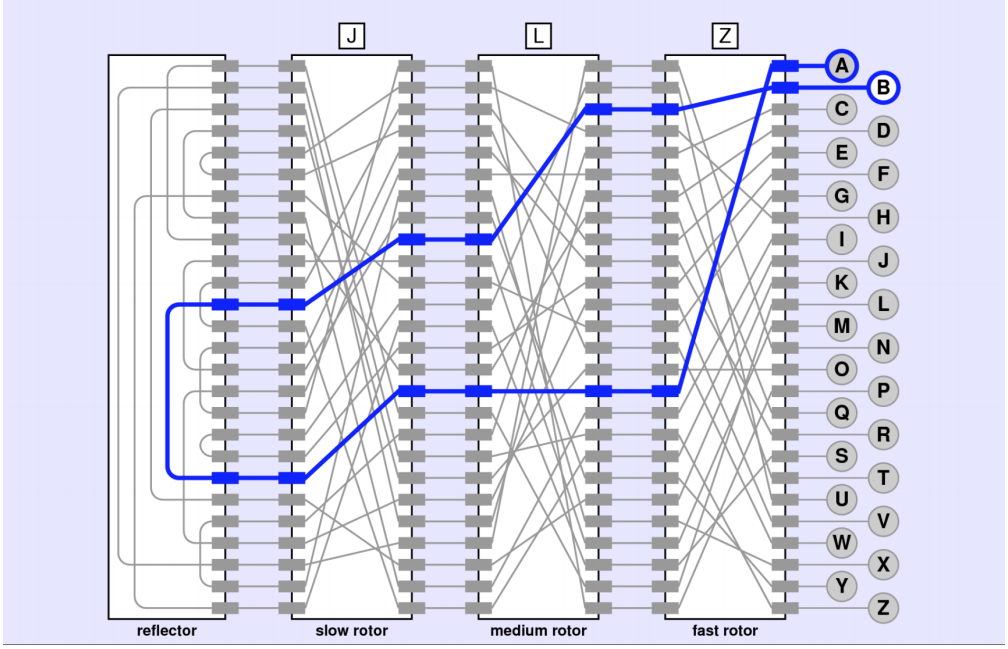


Figure 1: Enigma machine, pressing the key 'A' (credit Eric Roberts)

However, the characteristic that makes Enigma powerful is that after each such encrytption of a character, the first (*fast*) rotor rotates (as the name suggests) by one position, thus changing the transposition table. Moreover, when the fast rotor has completed a whole revolution, the *medium* rotor turns once and when the middle rotor completes a revolution, the *slow* rotor turns as well. As a result, pressing the same character twice does not yield the same output character (see figure 2).

Observe that thanks to the reflector, pressing the ciphered charcater's key when the machine is in the same setting as during encryption yields the original plaintext character! This makes the Enigma machine very convenient to use. The sender and the receiver only need to share the rotor positions when the encryption began. The receiver therefore

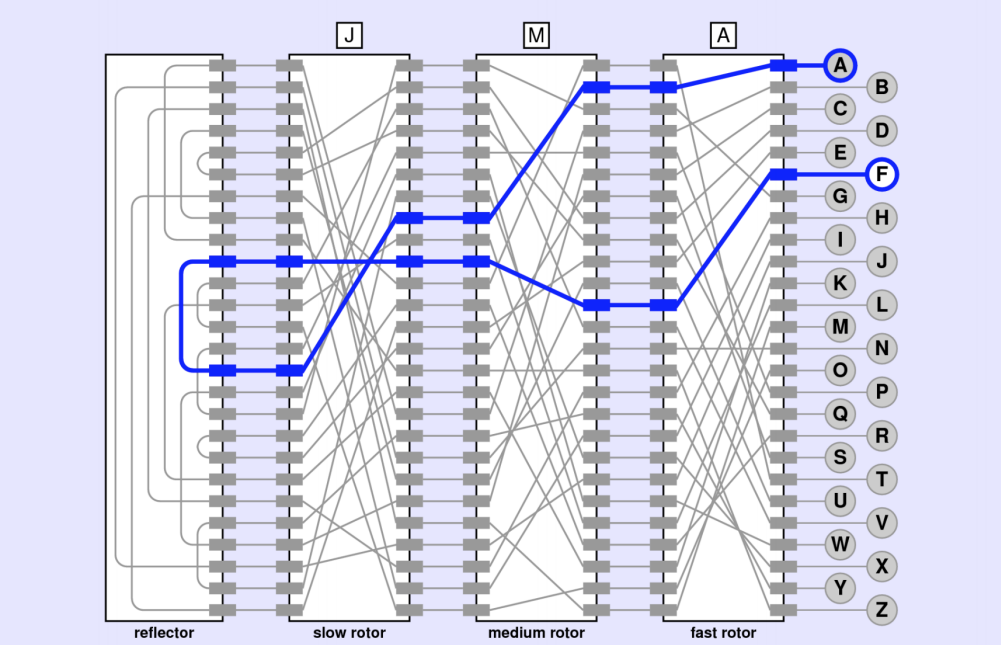


Figure 2: Enigma machine, pressing the key 'A' again (credit Eric Roberts)

simply types the ciphertext to obtain the plaintext. We call the positions of the rotors (in $\{0, \ldots, 25\}^3$) the *encryption key*, or simply the *key*.

Exercice 1 `singleRotorEmulate`**:** Write a constraint program which "emulates" a machine with a single rotor (and no reflector). The array of variables plaintext stands for the message to encrypt, ciphertext for the encrypted message and the variable key stands for the position of the rotor when encrypting the first character.

Exercice 2 `singleRotorBreak`**:** We want to decipher the encrypted message "LCKPYBKOUMPVHKZNBITLIKUXOVN" that was obtained by using a single rotor. We have a model of the rotor (`perm[0]`) that was used because our navy found it when boarding a U-pedal-boat, however we do not know what *key* was used to encrypt the message.

To give us the edge, our intelligence agency worked day and night and arrived to the following conclusion: the ennemy is devious, yet always very polite, so the message must start with a greeting ("HI" or "HELLO"). This is what is called a *crib*: we know that the letter 'H', when encoded in first position, corresponds to a 'L'.

Write a constraint program which decipher a message ("LCKPYBKOUMPVHKZNBITLIKUXOVN") encrypted with a single rotor (`perm[0]`), given a crib ("H").

Exercice 3 `multipleRotorsBreak`: We have intercepted a new ciphered message ("YTHXGIPIBUZYCCQSLPELWDRAOZXOSGOHCZNOLRGYXFYZTWFMZIBDBUVOU"), however, the ennemy has used not one but two rotors (a fast `perm[0]` and a medium rotor `perm[1]`). Being grossly arrogant, there are good reasons to believe that the ennemy sent a message starting with a demonstrative pronoum ("THE", "THAT", "THOSE", etc.), so the crib is "TH". Write a constraint program to decipher it.

Exercice 4 `commercialEnigmaEncode`: It appears that the machine that our navy found on the earlier U-pedal-boat, and which our expert analyst first classified as a "minitel" (whatever that is) was in fact an Enigma machine. With the knowledge of its mechanism (given by `rotors` and `reflector`), we will be able to break the ennemy code! First we should be able to emulate it. Write a constraint program which, given a message `plaintext` and an array of (3) rotor positions `key`, computes the encrypted message in the array of variables `ciphertext`.

Exercice 5 `commercialEnigmaBreak`: Alright, now we can eventually break the ennemy code! We managed to lure an ennemy homming pigeon by skilfully disguising a Sussex cottage into a Bavarian castle. Here's the message that was tied to its ankle: "SLIAMQPKAYMDNZPPJUWWWTNFCJZEHRNQOEANJGVBMHHDZBXMACJRZYPDXJUSWTGOPCJQUJCZVAMEDUPAIMPCXP". Our expert psychoanalysts tell us that the ennemy is so self-centered[1] that the message is likely to start with "WE". Write a constraint program to decipher it.

Breaking military Enigma[2] (`MilitaryEnigmaBreaker.java`)

Unfortunately, the Enigma machine in its military version was upgraded by the addition of a *plug board* (see figure 3). Before and after going through the rotors, the character signal is transposed using the chosen configuration of the plug board, which is a symmetric permutation. Only ten pairs of characters can be transposed, the remaining six characters are transposed to themselves. With this device, the encryption key now becomes the rotors' positions *plus* the plugboard pairings. This increases the number of potential keys from $17576 = 26^3$ to $150,738,274,937,250$...

Exercice 1 `militaryEnigmaBreak`: We will first try to generalise the model used to break the commercial version by adding variables for the plug board. However, since the problem is much harder we shall retrict the alphabet to 16 characters ("ACDEGHILMNORSTWY") and also restrict the plugboard so that it is only capable of transposing two pairs of characters (the remaining 22 characters are transposed to themselves). The message is [too long to state in the text] and the crib is "ALRIGHTTHISMESSAGEWAS". Write a constraint program to break it!

[1] the subsequent dispute about the root cause being mother or father-related hasn't stopped yet, but that's beside the point

[2] or trying to...

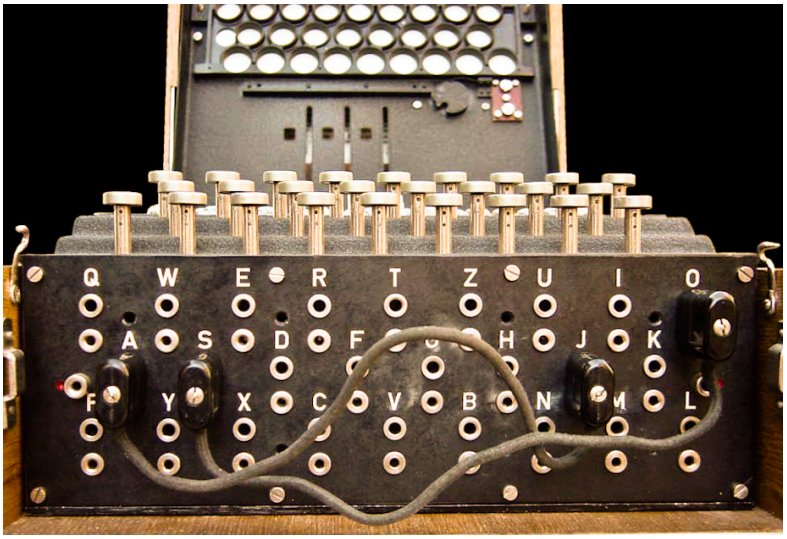


Figure 3: Enigma machine's plugboard (Wikimedia commons)

The goal here is to come up with the most efficient constraint program as possible. One way is to add *implied constraints*. An observation that might be useful is that the plugboard is static, it always transposes the same pairs of characters no matter where they are in the message. Another observation is that because the reflector is irreflective, Enigma never encrypts a character into itself.

Breaking military Enigma[3] (`Bombe.java`)

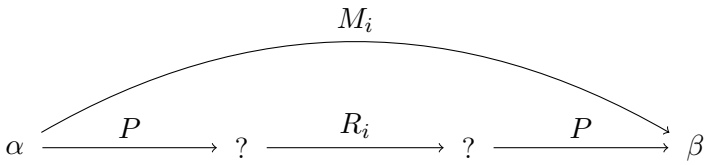
Unless you are an exceptionally good constraint programmer, your general Enigma-deciphering program will not scale to full alphabet and plugboard.

Thankfully, a certain Alan Turing came up with a device, the *Bombe* that can bring us closer to breaking the code. This device goes through every of the 26^3 starting rotor positions in turn and emulates the Enigma machine (skipping the plugboard) for the given key.

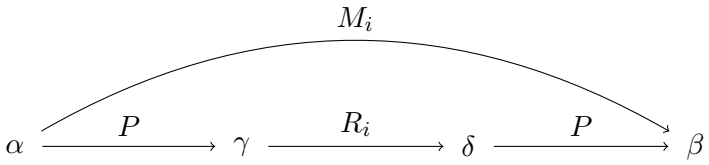
Let $M_i(\alpha)$ be the Enigma-encrypted character when reading α in i-th position the plaintext. Similarly, let $R_i(\alpha)$ be the image of α read i-th but skipping the plug board (forward and backward). Finally let $P(\alpha)$ be the image of α by the plugboard.

Since we have a crib, we do know some values of M_i, for instance suppose that at position i the plaintext is 'α' and the ciphertext is 'β' (i.e., $\beta = M_i(\alpha)$).

[3] for real this time



Moreover, let's pretend that the plugboard transposes α to γ. Since the bombe can emulate Enigma without plugboard, we can compute $R_i(\gamma)$, let it be δ.



We now know that if the plugboard transposes α to γ (or vice versa), then it must transpose δ to β (and vice versa): $P(\alpha) = \gamma \implies P(\delta) = \beta$. We can continue deducing transpositions and perhaps reach a contradiction (for instance it is a contradiction right away if $\delta = \alpha \neq \beta \neq \gamma$). We check every character of the alphabet in the same way.

When every character has at least one possible transposition, then the Bombe stops and the operator must search for every feasible transposition tables in order to find one that decipher the message.

Exercice 1 `BombeEnigmaBreak`: Alan Turing (who, though a bit geeky, is really a brilliant chap) observed that because the reflector is an irreflexive transposition, the Enigma machine *never* encrypts a character into itself. Our analysts think that the sentence "THEFIRSTONEWHOCAN" is somewhere in the message. Using Alan's trick, we can rule out a number of positions. For instance only the positions 1 and 4 are possible among the cases below:

```
  CAUBOJJOUTCTVHXQFJZUBZQCN
1:THEFIRSTONEWHOCAN
2:  THEFIRSTONEWHOCAN
3:   THEFIRSTONEWHOCAN
4:    THEFIRSTONEWHOCAN
5:     THEFIRSTONEWHOCAN
```

So we are going to try first with the crib being at the start of the message.

Adapt your constraint program for the military version of Enigma so that we can use it to find plugboard configurations when the Bombe stops. Instead of being an array of variables, now the key is an array of constant (input data).