Playfair

How to run the program

> ❗ **C++17 and CMAKE are required to run the code**

To build and run the program, run the following commands inside the project folder:

```
cmake .
make
./playfairCracker
```

What we found

Key: `MPQSKVWXZUHARLCENDTIFYBGO`

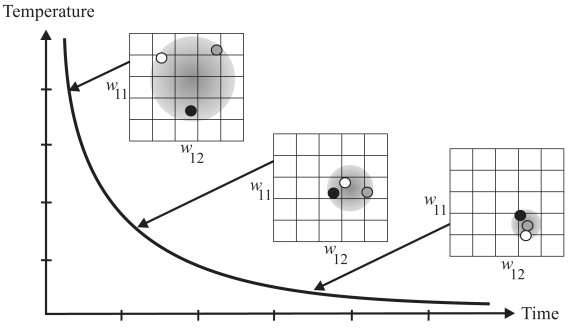
Plaintext: `INTHEBIGGESTBROWNESTMUDXDIESTRIVERINAFRICA…`

Approach used

Our Playfair cracker is based on the simulated annealing, a stochastic search approach inspired by the annealing process in metal work, where the key is randomly modified and the result is accepted if it improves the current score or if it meets a certain probability otherwise.

In fact also worse ones may also be accepted, based on a "temperature" parameter, which helps to prevent the algorithm from settling on local minima. Over time, the temperature reduces, lessening the acceptance of worse solutions and helping the algorithm converge to the best solution.

The score is calculated by a specific function about which we'll tell soon.



Ideal solution convergence

How it works

The program begins in the main:

```
int main()
    auto &cracker = Cracker::get_istance("ABCDEFGHIKLMNOPQRSTUVWXYZ","*****text to decipher*****");
    cracker.crack();
    std::cout << "Plaintext: " << cracker.get_text() << std::endl;
```

The cracker is launched and it takes in input the text to decipher and a starting key, the alphabet in our case.

```
__inline void crack() {
    srand(time(NULL));
    simulated_annealing();
    decrypt(m_key);
  }
```

The `crack()` method starts by initializing the random number generator with the current time.

Next, it runs the simulated annealing algorithm to find the best key to decrypt the ciphered text which will be used in the `decrypt(m_key)` method.

`simulated_annealing()`

The first step consist in a sort if initialization to set the current best score given the text and the random key we set at the beginning (the alphabet string).

Next, the method enters a loop that runs a number of iterations specified by `MAX_ITERATIONS` for each step of a temperature range specified by `ANNEALING_TEMP` and `ANNEALING_STEP_SIZE`. In each iteration, the method randomly modifies the current key using the `rearrange_key` function, decrypts the ciphered text with the new key, and calculates the score of the new decrypted text.

The `score_text` function iterates each quadgram and calculates the score of each one using the `score_quad` function. The function has to calculate the index of each character in the quadgram.

Finally, it returns the score of the quadgram from the `quadgram` vector, containing the score of all the possible quadgrams, using the sum of the character indices, which tells where to get the right value in the vector.

```
  for (double step = ANNEALING_TEMP; step >= 40; step -= ANNEALING_STEP_SIZE) {
    for (auto count = 0; count < MAX_ITERATIONS; count++) {
      new_key = key_copy;
      utils::rearrange_key(new_key);
      decrypt(new_key);
      score = utils::score_text(m_text);
      delta = score - max_score;
      if (delta >= 0) {
        max_score = score;
        key_copy = new_key;
      }
      else if (step > 0) {
        prob = exp(delta / step);
        if (prob > 1.0 * rand() / RAND_MAX) {
          max_score = score;
          key_copy = new_key;
        }
      }
      if (max_score > best_score) {
        best_score = max_score;
        m_key = key_copy;
        std::cout << "Attempt n: " << counter << " Key: " << m_key << " Score: " << best_score << std::endl;
      }
      counter++;
    }
  }
return best_score;
```

If the score of the new decrypted text is higher than the current score, the new key becomes the current key. If the score is lower, the new key can still become the current key with a certain probability that depends on the score difference and the current temperature. This allows the algorithm to explore the key space **even if the new key does not immediately improve the score.**

Finally, if the score of the current key is the best so far, the current key becomes the best key and the score is printed to the console. The method returns the score of the best key found.

`decrypt(m_key)`

With the possible key previously found, this method finally tries to decipher the given text reversing the procedure of encrypting with the Playfair rules we know.

It iterates over the string `m_ciphered`, containing the ciphered message. The iteration occurs in increments of 2, since in Playfair each pair of characters in `m_ciphered` represents a single character in the original message.

For each one, the method calculates the index of each character within the key. This is done using a lambda function `find_index` that finds the distance of the first occurrence of the character within the key from the beginning of the key.

The method calculates each character's row and column in a 5x5 grid by dividing the index by 5. Then it applies playfair encrypting rules to set the conditions: if characters share a row, they're replaced by their left grid characters. If they share a column, they're replaced by the characters above. In other cases, they're replaced by the intersection of their rows and columns. The result is saved in `m_text`.