

---

# *Soft Computing – 2006*

## *Cracking Codes With Genetic Algorithms*

Davide Eynard  
eynard@elet.polimi.it

Department of Electronics and Information  
Politecnico di Milano

# Perchè “Cracking codes”?

- Definizione sufficientemente ambigua per
  - ✓ **Cracking software** (code = ASM)
    - *Una volta compreso il funzionamento di un algoritmo, è possibile usare un GA per trovare automaticamente la soluzione desiderata*
    - *È necessario tener conto di questo quando si desidera **creare** una protezione software*
  - ✓ **Cracking ciphers** (code = cifratura)
    - *Le cifrature più semplici possono essere aperte in tempi ragionevoli con l'aiuto di un GA*

---

# Metodo

## **1) Scelta e studio del problema**

- *Scegliere un problema che sia possibile risolvere (tipologia – tempo – capacità – utilità)*
- *Studiarlo per poterlo comprendere al meglio*

## **2) Modello e cromosomi**

- *Corrispondenza fra cromosomi e soluzioni*

## **3) Fitness**

- *Valutare la bontà di ogni singolo cromosoma*

# Metodo

## **4) Riproduzione e mutazione**

- *Conservare le proprietà dei cromosomi*
- *Trasmettere le proprie qualità ai figli*

## **5) Parametri**

- *Dimensione della popolazione, probabilità di riproduzione, probabilità di mutazione*

## **6) Test e conclusioni**

- *Valutazione dei risultati e tweaking dei parametri*

# Netscape Cache Explorer

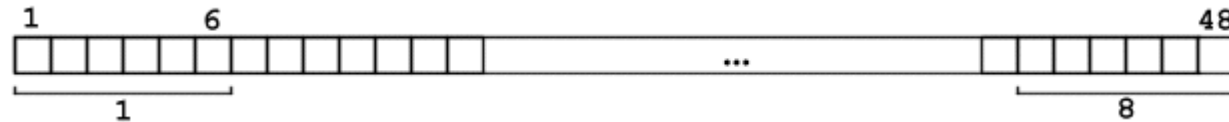
- ***nsce.exe*** by Matthias Wolf, shareware, v1.20, 22/08/1996
- Il programma chiede una chiave di registrazione  
<http://www.woodmann.com/fravia/nscekey.htm> [EN]  
<http://freeweb.supereva.com/underthenet/texts/nscekita.htm> [IT]
- Scopo: comprendere il funzionamento dell'algoritmo, adattarlo per un GA e usarlo per ottenere chiavi di registrazione corrette

## NSCE: l'algoritmo

- *username* e *password* di 8 caratteri
- $p_n$  = n-esimo elemento della password inserita -  
- ((n-esimo elemento della login)%26)+65) +  
+ (26, se il valore finale e' <0)
- $$\begin{array}{ll} r_1=p_1+0\%10 & r_5=p_5+4\%10 \\ r_2=p_2+1\%10 & r_6=p_6+5\%10 \\ r_3=p_3+2\%10 & r_7=p_7+6\%10 \\ r_4=p_4+3\%10 & r_8=p_8+7\%10 \end{array}$$
- $r_1*r_2 + r_2*r_3 + r_3*r_4 + r_4*r_5 + r_5*r_6 + r_6*r_7 + r_7*r_8 = 190h = 400d$

## NSCE: *modello e cromosoma*

- Ogni cromosoma deve rappresentare una stringa di registrazione *valida* (*!=corretta*): lunghezza di 8 byte, solo alcuni caratteri
- Stringa di bit => Password



I numeri da 1 a 8 sono usati come indici di  
`char *alpha = azAZ09-_` (set di 64 caratteri)

## NSCE: *fitness*

- Abbiamo un valore verso cui tendere:

$$r1*r2 + \dots + r7*r8 = 190h = 400d$$

- La *differenza* (in modulo)  $d$  fra il valore restituito dalla funzione di controllo e 400 può essere una buona stima
  - *Minimo* di  $d = \text{massimo}$  di  $-d$
  - Per non avere valori negativi, traslo la funzione verso l'alto  $(-d+400)$
  - In C: `(abs(400-abs(400-x)))`



## NSCE: riproduzione e mutazione

- Per il `GA1DBinaryStringGenome` le `GAlib` usano di default, rispettivamente,
  - `OnePointCrossover`
  - `FlipMutator`
- Altri operatori:
  - `UniformCrossover`
  - `EvenOddCrossover`
  - `TwoPointCrossover`

## NSCE: *parametri*

- **ps (*population size*)**
  - 10~50
- **pc (*prob. Crossover*)**
  - 0.9~0.99
- **pm (*prob. Mutazione*)**
  - 0.1
- **ng (*num. Generazioni*)**
  - Usato per il display (evoluzione piuttosto rapida)

## NSCE: conclusioni

- Funziona :)
  - Il programma riesce ad elaborare, in un tempo accettabile, chiavi sempre diverse e sempre valide
- Ha senso?
  - Trade off fra complessità dell'algoritmo e prestazioni del GA: questo era un caso semplice, ma se avessimo avuto un problema più complesso?
  - Insegnamento: **NO** alla security by obscurity!

## Cifrature monoalfabetiche (CM)

- Ad ogni lettera ne corrisponde un'altra (e una sola!)
  - Cifratura di Cesare, alfabeto mescolato, XOR (lavorando sui byte)
- Il caso più generale è quello in cui viene fornito l'*alfabeto cifrante*. Ad esempio:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

"Attacchiamo all'una"

ZYXWVITSRQPONMLKJIHGFEDCBA

"Zggzxxsrznl zoo'fmz"

- Risolvere la cifratura significa trovare l'alfabeto cifrante

## CM: *modello e cromosoma*

- La generica soluzione è un alfabeto cifrante
  - È sempre lungo 26 caratteri e contiene *tutte e sole* le lettere dell'alfabeto (è una sua *permutazione!*)
- Le GAlib forniscono gli strumenti per crearne uno
  - **GAStr**ing**Genome** consente di definire un cromosoma come una generica sequenza di alleli
  - Ogni allele è una lettera dell'alfabeto
  - Funzione di inizializzazione personalizzata

## CM: *fitness*

- Come valutare la bontà di un alfabeto?
  - Lo si usa per decifrare il testo e poi si verifica quanto “buono” è il testo stesso
- Tecniche:
  - Analisi delle frequenze
  - *Cluster* di vocali e consonanti
  - Digrammi e trigrammi
  - Dizionario

## CM: fitness

- <http://crank.sf.net>
  - È uno strumento *free* per la crittanalisi
  - *Riutilizzo di codice*: perchè reinventare l'acqua calda? Analisi delle frequenze, digrammi, trigrammi
- Calcolo della fitness:
  - 1) Il cromosoma è convertito in chiave per Crank
  - 2) Il testo viene decifrato con la chiave ottenuta
  - 3) Calcolo delle tabelle delle frequenze e errore
  - 4) Risultato:  $1 / (a * slft\_err + b * bift\_err + c * trift\_err)$

## CM: riproduzione e mutazione

- È necessario garantire le proprietà originali dei cromosomi
- Le GAlib offrono, rispettivamente:
  - `PartialMatchCrossover` (garantisce compatibilità, perdita di parte del patrimonio genetico)
  - `SwapMutator` (inverte alcuni alleli a caso)



## CM: *parametri*

- **ps:** 25~50
- **pc, pm:** di default, 0.9 e 0.01 (*poi modificati!*)
- **ng:** due modalità
  - numero massimo di generazioni (quando non si conosce il testo in chiaro)
  - valore ottimo di fitness (quando si è già risolto il problema: utile in fase di test)

## CM: *parametri*

- **text size:** trade off tra qualità e tempo
  - Tanto più lungo è il testo, tanto più precise sono le statistiche
  - Tanto più breve è il testo, tanto più veloce è il calcolo della fitness
  - Quattro versioni: completa, 8k, 3k, 1k
- **a, b, c:** sono i parametri della funzione
  - 1)  $1 / (a * slft\_err + b * bift\_err + c * trift\_err)$

## CM: test approfonditi

- Il GA non funziona sempre bene con i parametri di default

```
=====
Statistiche per ps=25 mg=3000 pm=0.01 pc=0.90
+-----+-----+
| Alfabeto cifrante          | Statistiche (medie su 10 esecuzioni) |
+-----+-----+
| zyxwvutsrqponmlkjihgfedcba | Score = 8.3881 Gen = 00870 Time = 015 |
| qrstuvwxyzabcdefghijklmnop | Score = 5.4116 Gen = 02650 Time = 042 |
| bqklhmnopjcarstufvdgwxzyi | Score = 4.9567 Gen = 02530 Time = 040 |
+-----+-----+

Legenda: ps = population size          | Score: punteggio medio
         mg = max generations           |         (8.7647 = alfabeto giusto)
         pm = probabilita' mutazione    | Gen   : numero medio di generazioni
         pc = probabilita' crossover    | Time  : tempo medio di esecuzione
=====
```

=> sono necessari dei *test* per tarare i parametri

## CM: *test approfonditi*

- L'eseguibile è stato modificato per accettare diversi *parametri*
- *Wrapper* in perl per il calcolo delle statistiche
- Dati dei test:
  - text size: 8k, ps=25, ng=5000
  - $a=1$ ,  $b=10$ ,  $c=100$
  - $pc=0.10\sim 0.90$  (ad intervalli di 0.10)
  - $pm=0.01, 0.02, 0.04, 0.08$

## CM: conclusioni

- Dai risultati dei test, possiamo trovare  $pc$  e  $pm$  ideali (*almeno a livello statistico*)
  - $pc=0.6$  e  $pm=0.04$
  - *Perchè* abbiamo migliori risultati abbassando  $pc$ ? Come cambia la convergenza cambiando i valori?
  - Non e' sufficiente avere uno score medio alto: verificare anche il numero medio di generazioni!
  - Più basse sono le probabilità, più veloce è l'algoritmo

## Bibliografia

- Darrel Whitley: *“A Genetic Algorithm Tutorial”*  
<http://citeseer.ist.psu.edu/29471.html>
- <http://galib.sourceforge.net>
- Ravi Ganesan, Alan T. Sherman:  
*“Statistical Techniques for Language Recognition: An Introduction and Guide for Cryptanalysts”*  
<http://citeseer.ist.psu.edu/ravi93statistical.html>
- <http://davide.eynard.it/malawiki/GeneticAlgos>