THE GAME

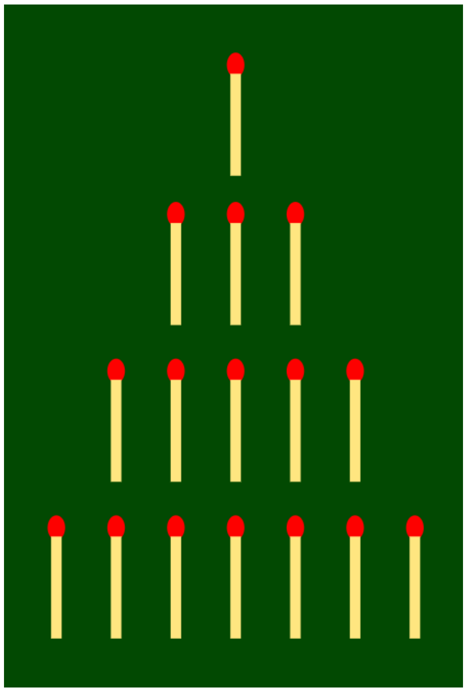


Figure 1: instance

STRATEGIES

```
  def strategy1(state:Nim)->Nimply:
  rows=state.rows
  chosen=np.argmax(rows)
  ply=Nimply(chosen,rows[chosen])
  return ply

  def optimal(state: Nim) -> Nimply:
    analysis = analize(state)
    logging.debug(f"analysis:\n{pformat(analysis)}")
    spicy_moves = [ply for ply, ns in analysis["possible_moves"].
    items() if ns != 0]
    if not spicy_moves:
        spicy_moves = list(analysis["possible_moves"].keys())
    ply = random.choice(spicy_moves)
    return ply
def strategy_random(state: Nim) -> Nimply:
    row = random.choice([r for r, c in enumerate(state.rows) if c >
     0])
    num_objects = random.randint(1, state.rows[row])
    return Nimply(row, num_objects)
```

STATE EMBEDDING

We used the *nimsum* function to describe the state we are into.

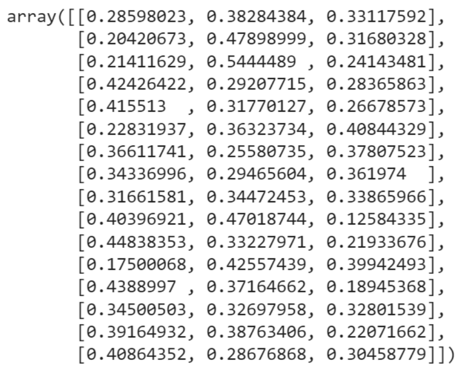
The *nimsum* is defined as follows:

```python
def nim_sum(state: Nim) -> int:
    tmp = np.array([tuple(int(x) for x in f"{c:032b}") for c in
    state.rows])
    xor = tmp.sum(axis=0) % 2
    return int("".join(str(_) for _ in xor), base=2)
```

So in our case where we have 5 rows it has 16 possible values.

INDIVIDUAL

With individual we imply a np.array with 16 rows and 3 columns where Ind[i,j]=probability of choosing strategy[j] if we are in state[i].



```
array([[0.28598023, 0.38284384, 0.33117592],
       [0.20420673, 0.47898999, 0.31680328],
       [0.21411629, 0.5444489 , 0.24143481],
       [0.42426422, 0.29207715, 0.28365863],
       [0.415513  , 0.31770127, 0.26678573],
       [0.22831937, 0.36323734, 0.40844329],
       [0.36611741, 0.25580735, 0.37807523],
       [0.34336996, 0.29465604, 0.361974  ],
       [0.31661581, 0.34472453, 0.33865966],
       [0.40396921, 0.47018744, 0.12584335],
       [0.44838353, 0.33227971, 0.21933676],
       [0.17500068, 0.42557439, 0.39942493],
       [0.4388997 , 0.37164662, 0.18945368],
       [0.34500503, 0.32697958, 0.32801539],
       [0.39164932, 0.38763406, 0.22071662],
       [0.40864352, 0.28676868, 0.30458779]])
```

Figure 2: instance

EVOLUTION

fitness

```python
def fitness(individual,accuracy):
  score=0
  i=1
  while i<=accuracy:
    score+=tournament(4,individual,strategies)
    i+=1
  return 1-score/accuracy
```

Basically the fitness function tells us the win rate of a given individual on the system of interest

parent selection

```python
def select_parent(pop,pressure):
  parent= random.choice(pop[:pressure])
  return parent

```

Given that the evolution algorithm presented always have population members sorted by performance, the pressure variable can be seen as how strict we are on the parent selection

mutation

```python
def mutate(individual,strategies,std):
  perturbations = np.random.normal(loc=0, scale=std, size=
    individual.shape)
  perturbations=np.abs(perturbations)
  mutated=individual+perturbations
  norms = np.sum(mutated, axis=1)
  for i in range(individual.shape[0]):
    mutated[i,:]/=norms[i]
  return mutated

```

fusion

```python
def fusion(p1,p2):
  o=(p1+p2)/2
  return o

```

BENCHMARK

The strategy to beat is

```python
def optimal(state: Nim) -> Nimply:
    analysis = analize(state)
    logging.debug(f"analysis:\n{pformat(analysis)}")
    spicy_moves = [ply for ply, ns in analysis["possible_moves"].
    items() if ns != 0]
    if not spicy_moves:
        spicy_moves = list(analysis["possible_moves"].keys())
    ply = random.choice(spicy_moves)
    return ply


```

EVOLUTION $(\mu + \lambda)$

We use a $(\mu + \lambda)$ strategy in which we :

GENERATE THE INITIAL POPULATION

```
population=list()
strategies=[strategy1,optimal,strategy_random]
for i in range(population_size):
  map=initialization(strategies)
  population.append(map)

```

GENERATE THE OFFSPRING

```
  offspring=list()
  for counter in range(off):
    if random.random()<mp:
      p=select_parent(population,pressure)  #tenere conto dei
    migliori
      o=mutate(p,strategies,10)
    else:
      p1=select_parent(population,pressure) #tenere conto dei
    migliori
      p2=select_parent(population,pressure)
      o=fusion(p1,p2)
    offspring.append(o)
  offspring.sort(key=lambda i:fitness(i,accuracy),reverse=True)
  off_best=fitness(offspring[0],accuracy)

```

Using mutation probability we decide whether pick a partner and mutate it or pick two of them and generate a son.

STEP

We sort the offsprings todecide whether it is a good set or not:

```
if off_best>=limit_pop:
    print("good offspring")
    population.extend(offspring)

```

limitpop is the limit(th) element of the sorted population,roughly speaking,the best of the offsprings has at least to be in top(limit).

With this part we can calibrate the algorithm and decide whether have it more or less aggressive.

ALGO

```
population.sort(key=lambda i:fitness(i,accuracy),reverse=True)
best_pop=fitness(population[0],accuracy)
limit_pop=fitness(population[limit],accuracy)
print(best_pop)

for generation in range(100):
  offspring=list()
  for counter in range(off):
    if random.random()<mp:
      p=select_parent(population,pressure)  #tenere conto dei
    migliori
      o=mutate(p,strategies,10)
    else:
      p1=select_parent(population,pressure) #tenere conto dei
    migliori
      p2=select_parent(population,pressure)
      o=fusion(p1,p2)
    offspring.append(o)
  offspring.sort(key=lambda i:fitness(i,accuracy),reverse=True)
  off_best=fitness(offspring[0],accuracy)
  if off_best>=limit_pop:
    print("good offspring")
    population.extend(offspring)
    population.sort(key=lambda i:fitness(i,accuracy),reverse=True)
    population=population[:population_size]
    best_pop=fitness(population[0],accuracy)
    limit_pop=fitness(population[limit],accuracy)
    print(best_pop)
  else:
    print("bad")

```