Simulated Annealing: 8 baby lizards problem

Optimization

Nieves Montes Gómez (1393150)

January 22, 2020

1 Introduction

In this assignment, the goal is to implement the Simulated Annealing (SA) algorithm to solve the 8 baby lizards problem, an analogous version to the benchmark 8-queens problem. In this case, attacks between lizards might be blocked by fixed trees.

SA is a suitable algorithm to solve this problem because it guarantees a global minimal, which is necessary in this case. It is often used for discrete problems with large search spaces. The first feature is clearly true in this case. In order to check that we are dealing with a large search space, we consider that to arrange 8 lizards in a $8 \times 8 = 64$ board with t trees blocking some positions, the number of possibilities is given by the following formula:

$$\binom{(64-t)}{8} = \frac{(64-t)!}{8! \cdot (56-t)!} \tag{1}$$

For a reasonable number of trees (t between 0 and 4) the above formula yields a search space of $\sim 10^9$ configurations.

Three cooling schedules have been investigated:

- Logarithmic cooling:
$$T_k = \frac{c}{\log(1+k)} \tag{2}$$
where k refers to the outer loop iteration, and c is a fixed constant.

- Geometric cooling:
$$T_{k+1} = \alpha \cdot T_k \tag{3}$$
where α is a fixed constant.

- Linear cooling:
$$T_{k+1} = T_k - \Delta T \tag{4}$$
where ΔT is also fixed. However, temperatures at or below 0 are not allowed. The minimum temperature of this schedule is 10^{-12}.

SA is implemented to find a solution to the problem for various board set-ups, with different numbers of trees. The influence of three cooling schedules presented will be investigated, as well as the effect of the Boltzmann constant.

2 Implementation

The complete Python code implementing SA for this problem can be found here. It it organized into three files. First, `configuration.py` contains the definition of `class Configuration()`, which represents a candidate solution:

```
class Configuration(object):
  """Class that represents a candidate configuration. Attributes are:
  - List of lizard positions. Initialized to empty list.
  - List of trees positions. Initialized from input, if any.
  - List of attacked lizards. All are initialized as NOT being attacked
  (False).
  - Energy of the configuration, initialized to 0."""
  def __init__(self, treeList=[]):
    self.lizardPositions = []
    if isinstance(treeList, list):
      self.treePositions = treeList
    else:
      self.treePositions = []
    self.attacked = [False for _ in range(8)]
    self.energy = 0
```

Its most important methods are:

- `def randomLizards(self)` places 8 lizards at random positions on the board, *i.e.* assigns a tuple of two integer numbers between 0 and 7 to each of them. All lizard positions are different to one another and to the positions of the trees (if any).
- `def isLizardAttacked(self)` scans the row, columns and diagonals within range of the l-th lizard to find out if there is any other baby that poses a threat. It returns a boolean value.
- `def findAttackedLizardsAndEnergy(self)` loops over all lizards to find out which ones are under attack. Then, it computes the energy of the configuration as the number of threatened lizards.
- `def plot(self)` produces a diagram of the board as those displayed in Figure 1.

Second, the file `coolingSchedule.py` includes the three different cooling schedules (see section 1) and an auxiliary function to plot the evolution of the algorithm (temperature and number of lizards under attack as a function of the outer loop iteration index).

Finally, file `SA.py` implements the actual algorithm. Its most important functions are the following:

```
  def NewConfiguration(config):
    """Takes in a configuration, and returns a new configuration created by
    moving an attacked lizard to new available position."""
    if not isinstance(config, configuration.Configuration):
      raise ValueError('Argument is not a configuration.')
    newConfig = copy.deepcopy(config)
    attackedArray = np.asarray(newConfig.attacked, dtype = np.bool8)
    attackedLizards = np.where(attackedArray == True)[0]
    if attackedLizards.size == 0:
      return newConfig
    randomIndex = int(os.urandom(1)[0]/255*attackedLizards.size - 1.E-9)
    movableLizard = attackedLizards[randomIndex]
    newPosition = (int(os.urandom(1)[0]/255*8 - 1.E-9), int(os.urandom(1)[0]/
                                  255*8 - 1.E-9))
    while (newPosition in newConfig.lizardPositions) or (newPosition in
                                  newConfig.treePositions):
      newPosition = (int(os.urandom(1)[0]/255*8 - 1.E-9), int(os.urandom(1)[0
                                  ]/255*8 - 1.E-9))
    newConfig.lizardPositions[movableLizard] = newPosition
    newConfig.findAttackedLizardsAndEnergy()
    return newConfig
```

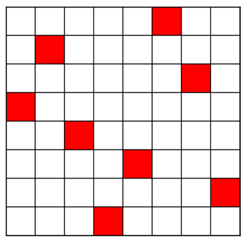
```
  def InnerLoop(config, temp):
    """Takes in a configuration and, for a fixed temperature, it looks for
    new configurations until either of this conditions is fulfilled:
    - The loop has exceeded a maximum number of iterations Lmax.
    - The loop has accepted a maximum number of new configurations Lamax.
    - The loop has found a new configuration with 0 energy, a solution to the
    problem.
    The function returns the last accepted configuration and the rate of
    accepted configurations."""
  if not isinstance(config, configuration.Configuration):
    raise ValueError('Argument is not a configuration.')
  L = 0 # total iterations
  La = 0 # total accepted iterations
  while (L < Lmax) and (La < Lamax) and (config.energy > 0):
    newConfig = NewConfiguration(config)
    if (newConfig.energy < config.energy):
      config = copy.deepcopy(newConfig)
      La += 1
    else:
      acceptProb = float(os.urandom(1)[0]/255)
      if acceptProb < math.exp((config.energy - newConfig.energy)/(kB*temp)):
        config = copy.deepcopy(newConfig)
        La += 1
    L += 1
  return (config, La/Lamax)
```

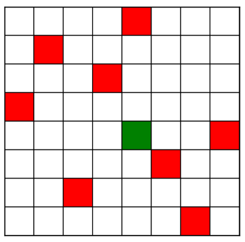
```
  def SimulatedAnnealing(treeList = [], cooling='geometric'):
    """Provided a treeList in the board, use Simulated Annealing to find a
    configuration of the lizards such that any of them is in danger. Start by
    creating an initial configuration with random lizard positions. The tree
    positions is taken from input. The cooling argument specifies the cooling
    schedule (geometric by default). The algorithm is only allowed to halt
    once a solution has been found."""
    if not isinstance(treeList, list):
      raise ValueError('Argument treeList is not a list.')
    initialConf = configuration.Configuration(treeList = treeList)
    initialConf.randomLizards()
    initialConf.findAttackedLizardsAndEnergy()

    # Find starting temperature
    temp = Tini
    r = InnerLoop(initialConf, temp)[1]
    while (r < HTsw):
      temp += dt
      r = InnerLoop(initialConf, temp)[1]
    print('Initial temperature: ' + str(temp))
    print('Lizards under attack: ' + str(initialConf.energy) + '\n')
    # SA - outer loop
    iterList = [0]
    lizUnderAttack = [initialConf.energy]
    tempList = [temp]
    candidate = initialConf
    outerIter = 1
    while (candidate.energy > 0) and (outerIter <= Lmax):
      candidate = InnerLoop(candidate, temp)[0]
      print('Iteration ' + str(outerIter))
      print('Temperature: ' + str(temp))
      print('Lizards under attack: ' + str(candidate.energy))
      print('\n')
      iterList.append(outerIter)
      lizUnderAttack.append(candidate.energy)
      tempList.append(temp)
      # update temperature according to schedule
      if cooling == 'geometric':
        temp = coolingSchedule.geometric(temp, alpha)
      elif cooling == 'logarithmic':
        temp = coolingSchedule.logarithmic(c, outerIter)
      elif cooling == 'linear':
        temp = coolingSchedule.linear(temp, deltaT)
      else:
        raise ValueError('Invalid cooling schedule.')
      outerIter += 1
    print('A solution has been found after ' + str(outerIter-1) + '
                                  iterations.')

    # roll back last temperature change
    if cooling == 'geometric':
      temp = temp/alpha
    elif cooling == 'logarithmic':
      temp = c / math.log()
    elif cooling == 'linear':
      temp += deltaT
    else:
      raise ValueError('Invalid cooling schedule.')

    print('Final temperature: ' + str(temp))
    return (candidate, iterList, lizUnderAttack, tempList)
```

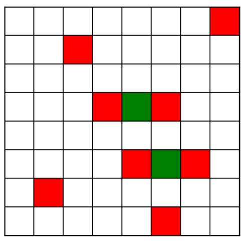
The important parameters related to the algorithm and the various cooling schedule are:

```
"""Parameters related to the SA algorithm"""
Lmax = 100 # maximum number of total inner loop iterations
Lamax = 10 # maximum number of accepted inner loop iterations
HTsw = 0.99 # minumum acceptance rate at starting temperature
Tini = 1.00 # initial temperature
dt = 0.01 # temperature increment to find starting temperature
kB = 1.0E-2 # Boltzmann constant

"""Parameters related to cooling schedule."""
alpha = 0.95 # geometric cooling
c = 0.30 # logarithmic cooling
deltaT = 0.01 # linear cooling
```

3 Results


(a)


(b)


(c)

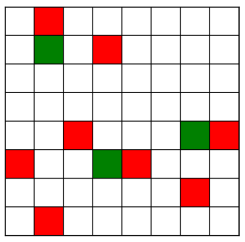
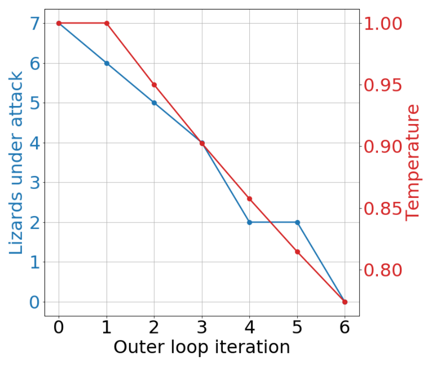
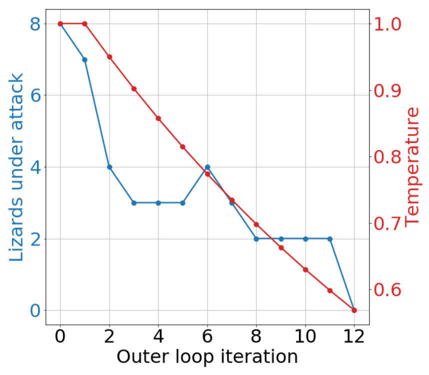

(d)

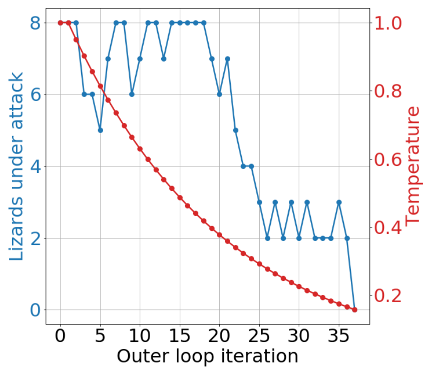
Figure 1: Example solutions obtained by the program for configurations with (a) zero, (b) one, (c) two and (d) three trees. Lizard positions are indicated by red tiles, trees by green tiles.



(a) `kB=1.0E-4`



(b) `kB=1.0E-2`



(c) `kB=1.0E0`

Figure 2: Effect of the Boltzmann constant `kB` on the evolution of the algorithm (number of lizards under attack and temperature versus outer loop iteration). The board set-up is fixed without trees, and the cooling schedule is geometric with `alpha=0.95`.

Some solutions for different board set-ups with increasing number of trees are displayed in Figure 1.

First, the effect of the Boltzmann constant `kB` is investigated. This parameter is related to how easily the algorithm accepts configurations with more attacked lizards than the current one. The results of a test for a fixed board set-up (no tress) and cooling schedule (geometric with $\alpha = 0.95$) are displayed in Figure 2. The evolution of the algorithm is plotted for increasing values of `kB`.

This test shows that for sufficiently small values of `kB` ($\sim 10^{-4}$), the probability of accepting energy-increasing configurations is virtually zero. If lucky, this will lead to a solution in very few iterations. However, it has been observed that if the system may get trapped in a non-optimal configuration that cannot access less energetic configurations, so the outer SA loop become infinite.

On the other end, a large `kB` (~ 1) easily accepts worse candidates, even as the temperature drops. In general, SA now needs many iterations to find a solution. An intermediate value for `kB` ($\sim 10^{-2}$) is considered to be a good compromise between the two extremes. This is the choice for the Boltzmann constant in the subsequent runs.

Table 1: Average number of iterations before the algorithm halts for each cooling schedule (including the characteristic parameter) and various number of trees in the board, taken over 100 runs of the algorithm. In all the runs, the program always finds a zero-energy solution. The tree positions correspond to those displayed in Figure 1.

Cooling schedule	**Parameter**	**Board set-up**			
		0 trees	**1 tree**	**2 trees**	**3 trees**
Logarithmic	$c = 0.30$	20.29	7.92	5.51	5.31
Geometric	$\alpha = 0.95$	15.69	7.08	5.98	5.08
Linear	$\Delta T = 0.01$	14.98	6.86	5.89	5.10

A comparative analysis on both the effect of the board set-up and the cooling schedule is presented in Table 1. For a fixed temperature evolution regime, the average number of iterations decreases as more trees are introduced into the terrarium. This result is expected, as trees block attacks and facilitate finding safe positions.

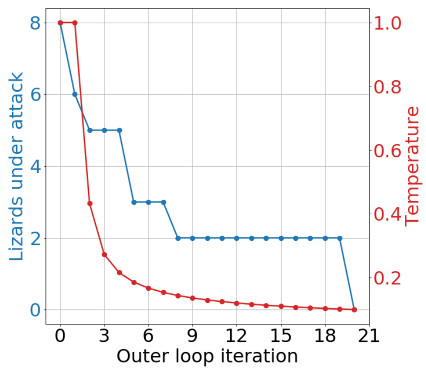
Going from a tree-free board to one tree makes a big difference, however the introduction of a new one is less impacting when many plants are already present. This is also somewhat intuitive, as a new tree might just be re-blocking already impeded directions.

When fixing the number of trees and comparing the three cooling schedules, the most noticeable difference is found for the 0-trees set-up. This leads to the first conclusion that the presence of trees dominates over the choice of cooling schedule.

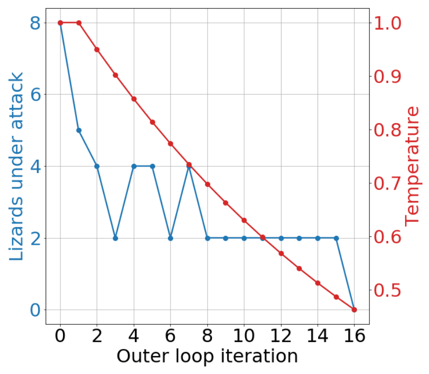
Precisely for this configuration, the evolution of SA for the three schedules is shown in Figure 3. Program runs that halt in a number of iteration close to the averages presented in Table 1 have been purposefully selected.

The logarithmic schedule (Figure 3a) decays very rapidly initially, but then stabilizes at a relatively low temperature, below 0.2. This is the lowest temperature reached by any of the cooling schedules. Under these conditions, progress stagnates for many iterations. This rapid initial decrease of temperature, which drops the chances of accepting more energetic configurations and hence reduces the explorability of the algorithm, is possibly the responsible for the "large" number of iterations needed to find a solution.

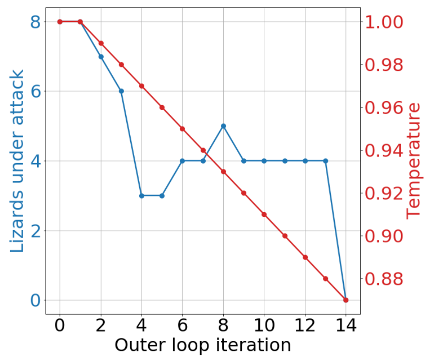
The geometric (Figure 3b) and linear (Figure 3c) cooling schedules have somewhat a similar evolution, since for a small number of iterations (as is the case) the exponential decay can be approximated by a linear function. However, at each step the geometric schedule reaches a much lower temperature than the linear one does.



(a) `cooling='logarithmic'`



(b) `cooling='geometric'`



(c) `cooling='linear'`

Figure 3: Evolution of the algorithm for the different cooling schedules (parameters have the same values as in Table 1) for a board with no trees (like that displayed in Figure 1a). Runs that end in a number of iterations close to the average have been selected. However, large variations exist between different runs with the same cooling schedule.

4 Conclusions

In this assignment, Simulated Annealing has been implemented to solve a variant of the benchmark 8-queens problem. First, the effect of the Boltzmann constant has been investigated. It has been established that the best choice is an intermediate value that allows for accepting some energy increases, but not so many as to hinder the progress of the search.

Then, three typical cooling schedules have been compared: logarithmic, geometric and linear. It is concluded that the last two (geometric with $\alpha = 0.95$ or linear with $\Delta T = 0.01$) are the preferred choices for running the algorithm, as they solve the problem in the least number of iterations for the most challenging set-up (empty board with no trees). However, the selection for the cooling schedule becomes less relevant as more protecting plants are introduced.

The problem proposed in this exercise could also have been solved with Genetic Algorithms (GAs). However, this is not as recommended as Simulated Annealing. To start, GAs need to store in memory many configurations at a time, enough to represent a whole population. By contrast, SA only needs to simultaneously store the current configuration and the possible next candidate.

Also, given the characteristics of the problem, Simulated Annealing makes more sense than Genetic Algorithms from the search point of view. For example, let's suppose that at the current state of the search, there is a very promising configuration where only two lizards are attacking each other. The SA procedure will make only a change to one of the lizard positions. If lucky, this may conclude the search.

Now, let's suppose that this configuration is an exceptionally fit individual among all the population in a GA. Despite having a large probability of being selected for breeding, its offspring will inherent only half of its genetic material, which might completely ruin its low energy positions. Therefore, for this particular problem, Simulated Annealing is considered a better approach.