IART - Artificial Intelligence

Exercise 4: Optimization/Meta-Heuristics

Luís Paulo Reis

LIACC – Artificial Intelligence and Computer Science Lab.
DEI/FEUP – Informatics Engineering Department, Faculty of Engineering of the University of Porto, Portugal
APPIA – Portuguese Association for Artificial Intelligence

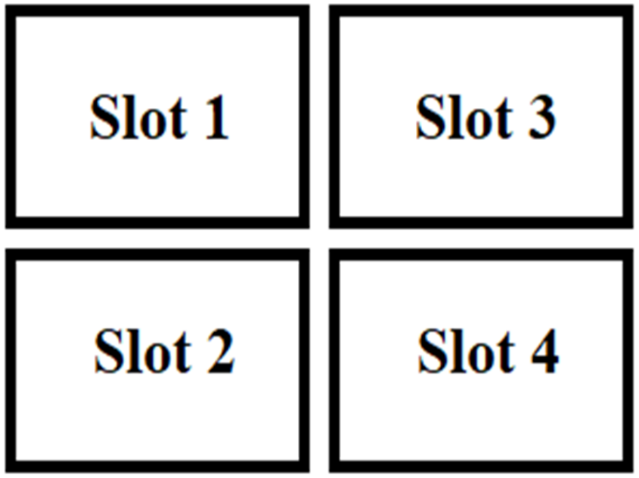
Exercise 4: Timetabling Problem Solving using Local Search/Simulated Annealing

Suppose that you have access to the information on students enrolled in various elective subjects of a master's/doctoral program. Each subject has only one weekly class.

We want to build a schedule/timetable using only a specified number of slots (available hours) and minimizing the number of incompatibilities for students (i.e., subjects that enrolled students will not be able to attend because of temporal overlaps in the respective schedules).

To do this, the class of each subject must be assigned to a slot (from the available ones).

Consider the example problem represented in the next slide.



Exercise 4.1: Timetabling Problem Solving using Local Search/Simulated Annealing

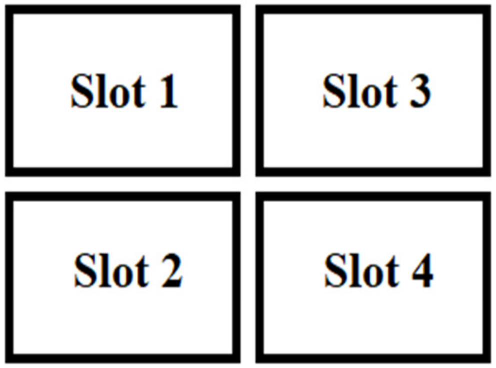
Data Representation

Facts Representation

```
slots (4).
disciplines (12).
students (12).
discipline (1, [1,2,3,4,5]).  % Students 1,2,3,4,5
                              % enrolled in discipline 1
discipline (2, [6,7,8,9]).
discipline (3, [10,11,12]).
discipline (4, [1,2,3,4]).
discipline (5, [5,6,7,8]).
discipline (6, [9,10,11,12]).
discipline (7, [1,2,3,5]).
discipline (8, [6,7,8]).
discipline (9, [4,9,10,11,12]).
discipline (10, [1,2,4,5]).
discipline (11, [3,6,7,8]).
discipline (12, [9,10,11,12]).   %Students 9,10,11,12
                                 % enrolled in discipline 12
```

Text File Representation

```
4 12 12      // Problem with 4 Slots and 12
             // Disciplines and 12 students
1 2 3 4 5    // Students 1,2,3,4,5 enrolled in discipline 1
6 7 8 9
10 11 12
1 2 3 4
5 6 7 8
9 10 11 12
1 2 3 5
6 7 8
4 9 10 11 12
1 2 4 5
3 6 7 8
9 10 11 12    // Students 9,10,11,12
              // enrolled in discipline 12
```



Exercise 4.1: Timetabling Problem Solving using Local Search/Simulated Annealing

a) Define a means of representation of a solution schedule and create a method that allows you to randomly create a solution.

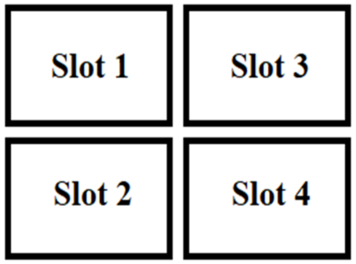
b) Build a function (on a language of your choice) that allows you to calculate an incompatibilities table, that is, for each pair of subjects (discipline) calculates the number of students who are enrolled in both.

c) In this problem, the simplest representation for a solution (assignment of disciplines to slots) consists of associating, to each discipline 1..*nd*, a slot 1..*ns*, where *nd* is the number of disciplines and *ns* is the number of slots. For this purpose, we can use a list of integers *nd*, whose values are numbers from 1 to *ns*. The list index identifies the respective discipline, and the value inside the list represents the slot to which it has been assigned. For example, in list [4,1,2,3,2,4,1,1,2,2,2,3] discipline 1 has been assigned to slot 4, such as discipline 6. Implement a function that allows you to evaluate a particular solution, by calculating the total number of students enrolled in overlapping subjects.

d) Define one or more neighboring spaces and neighbor functions capable of calculating the neighbors of a solution.

e) Build a program that allows you to use the following methods to find the optimal (or sub-optimal) solution to this problem:

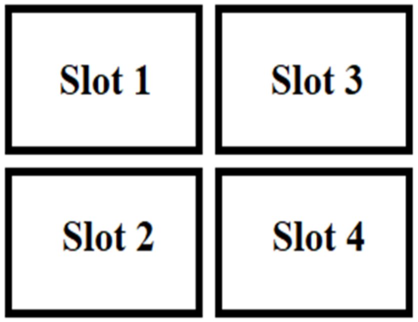
- Hill Climbing (multiple versions)
- Simulated Annealing



Exercise 4.1: Timetabling Problem Solving using Local Search/Simulated Annealing

f) Consider the following initial solutions and try the various methods developed:

- initial([1,1,1,1,1,1,1,1,1,1,1,1]).   % Solution with equal values
- initial ([1,1,1,2,2,2,3,3,3,4,4,1]).   % Almost the solution?
- initial ([1,1,4,2,2,2,3,3,3,4,4,1]).   % Local Minimum?
- initial ([1,2,3,4,1,2,3,4,1,2,3,4]). % Local minimum?



g) Try other initial solutions; use a random initial solution and the "Random Restarts" method.

h) Compare the implemented methods in terms of the quality of the obtained solution and the time it takes to obtain the solution, from the examples given.

f) Try creating several instances of the problem, with different dimensions (varying the number of disciplines and slots) and difficulties (varying the students enrolled in the disciplines).

Solving a Timetabling Problem

a) Define a means of representation of a solution schedule and create a method that allows you to randomly create a solution.

- **Solution Representation?**

```
def initial_solution():
```

Solving a Timetabling Problem

a) Define a means of representation of a solution schedule and create a method that allows you to randomly create a solution.

- **Solution Representation:**
  - 12 variables {Sol1, Sol2, … Sol12} with values 1..4
  - Array with 12 positions/disciplines: int Sol[12] with values 1..4 (slots)

```
def initial_solution():
```

Solving a Timetabling Problem

a) Define a means of representation of a solution schedule and create a method that allows you to randomly create a solution.

- **Solution Representation:**
  - 12 variables {Sol1, Sol2, … Sol12} with values 1..4
  - Array with 12 positions/disciplines: Sol[12] with values 1..4 (slots)

```
def initial_solution():
    sol=np.random.randint(1,slots+1,disc)
    #sol=[1,1,1,1,1,1,1,1,1,1,1,1]
    #sol=[1,1,1,2,2,2,3,3,3,4,4,1]
    #sol=[1,1,4,2,2,2,3,3,3,4,4,1]
    #sol=[1,2,3,4,1,2,3,4,1,2,3,4]
    #sol=[3,3,3,4,4,2,1,1,1,2,2,4]
    return sol
```

Solving a Timetabling Problem

a) Define a means of representation of a solution schedule and create a method that allows you to randomly create a solution.

- **Solution Representation:**
  - 12 variables {Sol1, Sol2, … Sol12} with values 1..4
  - Array with 12 positions/disciplines: Sol[12] with values 1..4 (slots)

- **Optimal Solution:**

  [1,1,1,2,2,2,3,3,3,4,4,4] - Evaluation = 0 (no incompatibilities)

  Also all 16 permutations… Example: [2,2,2,1,1,1,3,3,3,4,4,4]

  Probably other optimal solutions?

State Space

- **What is the size of the Problem State Space?**

State Space

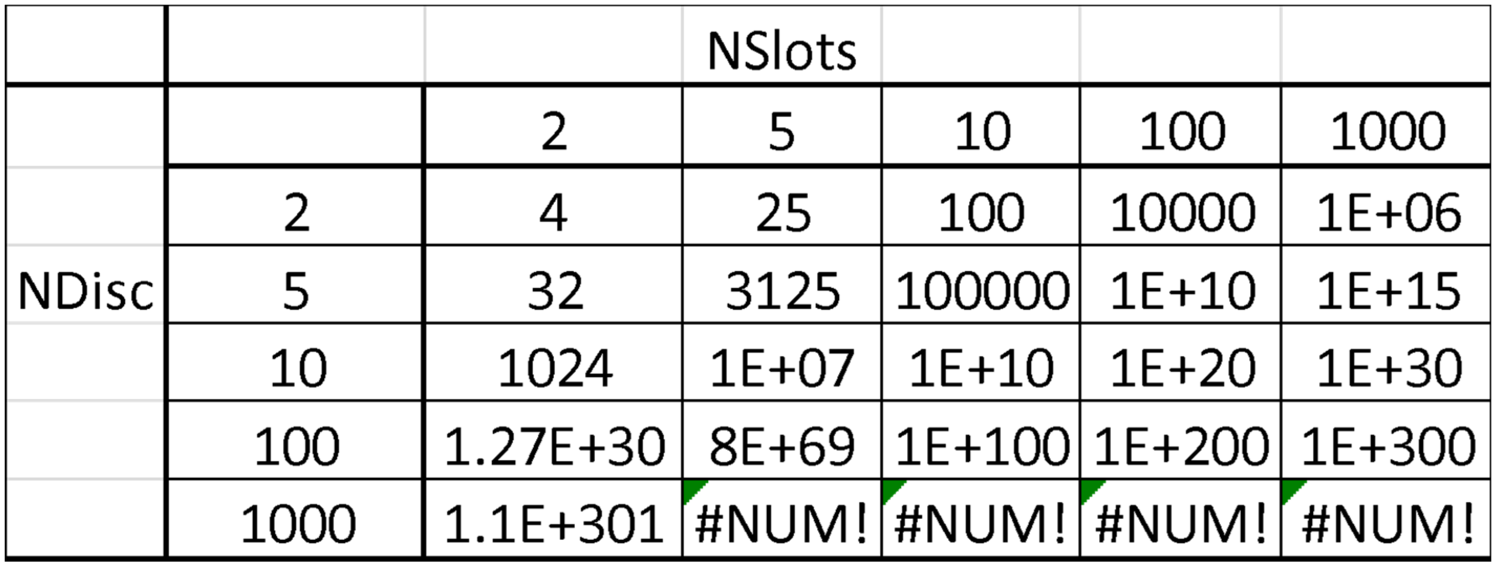
- **What is the size of the Problem State Space?**

- **Considering 12 disciplines and 4 slots**
  - State Space = 4^{12} = 16777216
- **Considering Nd disciplines and Ns slots**
  - State Space = Ns^{Nd}

State Space

- **What is the size of the Problem State Space?**

- **Considering 12 disciplines and 4 slots**
  - State Space = 4^{12} = 16777216
- **Considering Nd disciplines and Ns slots**
  - State Space =Ns^{Nd}



				NSlots		
		2	5	10	100	1000
	2	4	25	100	10000	1E+06
NDisc	5	32	3125	100000	1E+10	1E+15
	10	1024	1E+07	1E+10	1E+20	1E+30
	100	1.27E+30	8E+69	1E+100	1E+200	1E+300
	1000	1.1E+301	#NUM!	#NUM!	#NUM!	#NUM!

Solving a Timetabling Problem

b) Build a function (on a language of your choice) that allows you to calculate an incompatibilities table, that is, for each pair of subjects (disciplines) calculates the number of students who are enrolled in both.

Data is read to matrix: bool discStud [Nd, Na]

// discStud[4,8] = True => Student 8 registred at discipline 4

```
def imcomp(d1,d2):
…
      return count
```

Solving a Timetabling Problem

b) Build a function (on a language of your choice) that allows you to calculate an incompatibilities table, that is, for each pair of subjects (disciplines) calculates the number of students who are enrolled in both.

Data is read to matrix: bool discStud [Nd, Na]

// discStud[4,8] = True => Student 8 registed at discipline 4

```
def enroled (std,disc):
    check_disc = distr[disc]
    return (std in check_disc)

def incomp(d1,d2):
    count = 0
    for stud in range (1,std+1):
        if enroled(stud,d1) and enroled(stud,d2):
            count+=1
    return count
```

Solving a Timetabling Problem

In this problem, the simplest representation for a solution (assignment of disciplines to slots) consists of associating, to each discipline 1..nd, a slot 1..ns, where nd is the number of disciplines and ns is the number of slots. For this purpose, we can use a list of integers nd, whose values are numbers from 1 to ns. The list index identifies the respective discipline, and the value inside the list represents the slot to which it has been assigned. For example, in list [4,1,2,3,2,4,1,1,2,2,2,3] discipline 1 has been assigned to slot 4, such as discipline 6.

c) Implement a function that allows you to evaluate a particular solution, by calculating the total number of students enrolled in overlapping subjects.

```
def evaluate(solu):
    evals = 0
    …
    return evals
```

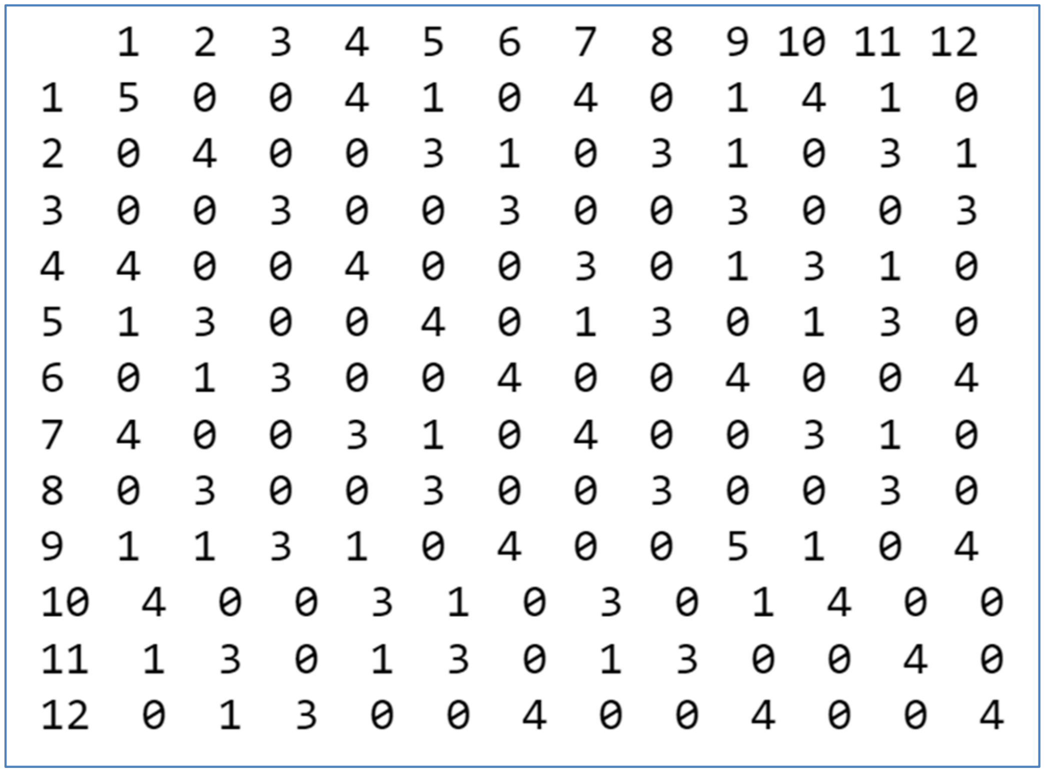
Solving a Timetabling Problem

c) Implement a function that allows you to evaluate a particular solution, by calculating the total number of students enrolled in overlapping subjects.

```
def evaluate(solu):
    evals = 0
    for d1 in range(0,disc-1):
        for d2 in range (d1+1,disc):
            if solu[d1] == solu[d2]:
                evals += imcomp(d1,d2)
    return evals
```

Test Incompatibility Matrix

```
#Test Incompatibility Matrix
print('    1   2   3   4   5   6   7   8   9 10 11 12')
for d1 in range(0,disc):
        print(d1+1, end='   ')
        for d2 in range (0,disc):
            print(imcomp(d1,d2), end='   ')
        print()
```



```
    1   2   3   4   5   6   7   8   9  10  11  12
1   5   0   0   4   1   0   4   0   1   4   1   0
2   0   4   0   0   3   1   0   3   1   0   3   1
3   0   0   3   0   0   3   0   0   3   0   0   3
4   4   0   0   4   0   0   3   0   1   3   1   0
5   1   3   0   0   4   0   1   3   0   1   3   0
6   0   1   3   0   0   4   0   0   4   0   0   4
7   4   0   0   3   1   0   4   0   0   3   1   0
8   0   3   0   0   3   0   0   3   0   0   3   0
9   1   1   3   1   0   4   0   0   5   1   0   4
10   4   0   0   3   1   0   3   0   1   4   0   0
11   1   3   0   1   3   0   1   3   0   0   4   0
12   0   1   3   0   0   4   0   0   4   0   0   4
```

Solving a Timetabling Problem

d) Define one or more neighbouring spaces and neighbour functions capable of calculating the neighbours of a solution.

Example of Sol = [1,1,1,1,2,2,2,2,3,3,4,4]

Possible Neighbours:

[3,1,1,1,2,2,2,2,3,3,4,4],

[1,1,1,1,2,2,4,2,3,3,4,4],

…

Solving a Timetabling Problem

d) Define one or more neighbouring spaces and neighbour functions capable of calculating the neighbours of a solution.

Example of Sol = [1,1,1,1,2,2,2,2,3,3,4,4]

// Neighbour1: Change a discipline from slot

Possible Neighbours:

[3,1,1,1,2,2,2,2,3,3,4,4],

[1,1,1,1,2,2,4,2,3,3,4,4],

…

```
def neighbor1 (solu):
    …
    return neig
```

Solving a Timetabling Problem

d) Define one or more neighbouring spaces and neighbour functions capable of calculating the neighbours of a solution.

Example of Sol = [1,1,1,1,2,2,2,2,3,3,4,4]

// Neighbour1: Change a discipline from slot

Possible Neighbours:

[3,1,1,1,2,2,2,2,3,3,4,4],

[1,1,1,1,2,2,4,2,3,3,4,4],

…

# Not the best solution! Think in a better one!

```
def neighbor1 (solu):
    neig = copy.deepcopy(solu)
    d1 = np.random.randint(0,disc)
    old = neig[d1]
    while True:
        new = np.random.randint(1,slots+1)
        if new!=old: break
    neig[d1] = new
    return neig
```

Solving a Timetabling Problem

d) Define one or more neighbouring spaces and neighbour functions capable of calculating the neighbours of a solution.

Example of Sol = [1,1,1,1,2,2,2,2,3,3,4,4]

// Neighbour 2: Exchange the slots of two disciplines

Possible Neighbours (exchange):

[1,1,1,1,2,2,2,2,3,4,3,4], [1,1,1,2,1,2,2,2,3,3,4,4], …

# Not the best solution! Think in a better one!

```
def neighbor2(solu):
    neig = copy.deepcopy(solu)
    d1 = np.random.randint(0,disc)
    n = 0
    while n<100:
        d2 = np.random.randint(0,disc)
        if d1!=d2 and neig[d1]!=neig[d2]: break
        n += 1;
    aux = neig[d1]; neig[d1] = neig[d2]; neig[d2] = aux
    return neig
```

Solving a Timetabling Problem

d) Define one or more neighbouring spaces and neighbour functions capable of calculating the neighbours of a solution.

// Neighbour3: Change a discipline from slot or exchange the slots of two disciplines

Possible Neighbours:

[3,1,1,1,2,2,2,2,3,3,4,4], [1,1,1,1,2,2,4,2,3,3,4,4], …

[1,1,1,1,2,2,2,2,3,4,3,4], [1,1,1,2,1,2,2,2,3,3,4,4], …

```
def neighbor3 (solu):
    if np.random.randint(0,2)==0:
        return neighbor1(solu)
    else:
        return neighbor2(solu)
```

Neighbourhood Tests

```
# Neighbourhood Tests

sol=np.random.randint(1,slots+1,disc)
print(sol)
for d in range(1,11):
    print(sol, neighbor1(sol))
print(sol)
for d in range(1,11):
    print(sol, neighbor2(sol))
```

Solving a Timetabling Problem

e) Build a program that allows you to use the following methods to find the optimal (or sub-optimal) solution to this problem:

- Hill Climbing (random neighbour)
- Simulated Annealing (with colling schedule: $T_{i+1} = 0.9T_i$)

Note: Consider a stop_criteria of 1000 iterations without improvement.

```
def hillclimbing():
      it = 0 ; itNoImp = 0
      sol = initial_solution()
      …
      while itNoImp < 10000:
            …
      return sol
```

Solving a Timetabling Problem

e) Build a program that allows you to use the following methods to find the optimal (or sub-optimal) solution to this problem:

· Hill Climbing (random neighbour)

Note: Consider a stop_criteria of 1000 iterations without improvement.

```
def hillclimbing():
    it = 0 ; itNoImp = 0
    sol = initial_solution()
    sol_eval = evaluate(sol)
    while itNoImp < 10000:
        it += 1; itNoImp += 1
        neig = neighbor3(sol)
        neig_eval = evaluate(neig)
        if (neig_eval < sol_eval):
            sol_eval = neig_eval
            sol = copy.deepcopy(neig)
            itNoImp = 0
    return sol
```

Solving a Timetabling Problem

e) Build a program that allows you to use the following methods to find the optimal (or sub-optimal) solution to this problem:

· Hill Climbing (random neighbour)

Note: Consider a stop_criteria of 1000 iterations without improvement.

```
def hillclimbing():
    it = 0 ; itNoImp = 0
    sol = initial_solution()
    sol_eval = evaluate(sol)
    #print('initial eval',evaluate(sol))
    while itNoImp < 10000:
        it += 1; itNoImp += 1
        neig = neighbor3(sol)
        neig_eval = evaluate(neig)
        #print(it, ') ',sol, ' -> ', neig, ':', neig_eval )
        if (neig_eval < sol_eval):
            sol_eval = neig_eval
            sol = copy.deepcopy(neig)
            #print(it, ') Solution - ',sol, ":", sol_eval )
            itNoImp = 0
    return sol
```

Solving a Timetabling Problem

e) Build a program that allows you to use the following methods to find the optimal (or sub-optimal) solution to this problem:

· Simulated Annealing (with colling schedule: $T_{i+1} = 0.99T_i$)

Note: Consider a stop_criteria of 1000 iterations without improvement.

```
def simulatedAnnealing():
    it = 0 ; itNoImp = 0
    T = 1000
    sol = initial_solution()
    while itNoImp < 10000:
        T = T_schedule(T)
        …
        neig = neighbor3(sol)
        …
    return solF
```

Solving a Timetabling Problem

e) Build a program that allows you to use the following methods to find the optimal (or sub-optimal) solution to this problem:

· Simulated Annealing (with colling schedule: $T_{i+1} = 0.99T_i$)

Note: Consider a stop_criteria of 1000 iterations without improvement.

```
def simulatedAnnealing():
    it = 0 ; itNoImp = 0 ; T = 1000
    sol = initial_solution(); sol_eval = evaluate(sol)
    solF = sol; solF_eval = sol_eval
    while itNoImp < 10000:
        T = T_schedule(T)
        it += 1; itNoImp += 1
        neig = neighbor3(sol)
        neig_eval = evaluate(neig)
        delta =sol_eval-neig_eval
        if delta>0 or np.exp(delta/T)>random.random():
            sol = copy.deepcopy(neig); sol_eval = neig_eval
            if (sol_eval < solF_eval):
                    solF = sol; solF_eval = sol_eval
                    itNoImp = 0
    return solF
```

Solving a Timetabling Problem

- **Simulated Annealing**

		T						
		0.001	1	2	4	10	100	1000
	0	100%	100%	100%	100%	100%	100%	100%
	1	0%	37%	61%	78%	90%	99%	100%
	2	0%	14%	37%	61%	82%	98%	100%
Delta	4	0%	2%	14%	37%	67%	96%	100%
	10	0%	0%	1%	8%	37%	90%	99%
	100	0%	0%	0%	0%	0%	37%	90%
	1000	0%	0%	0%	0%	0%	0%	37%

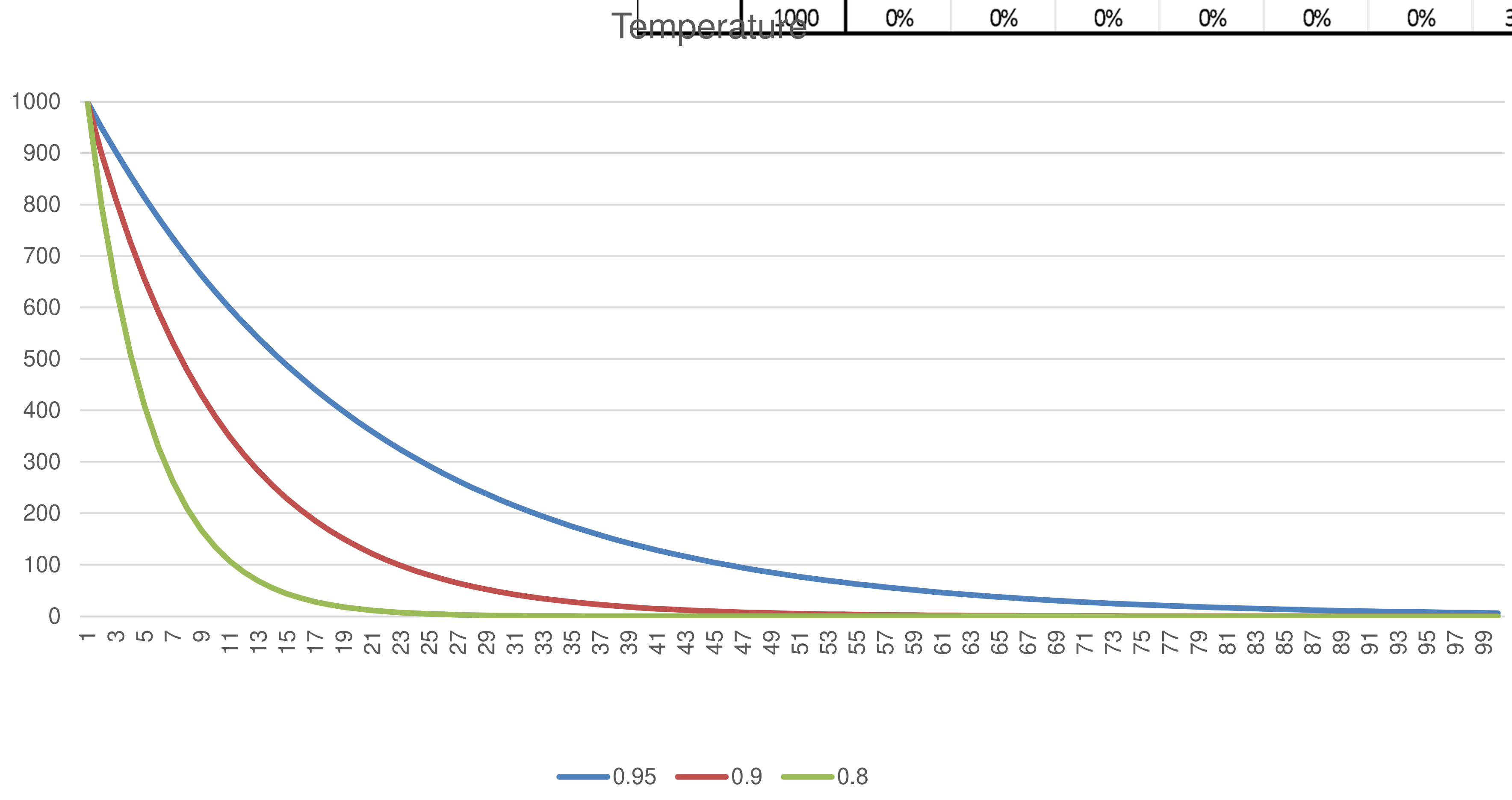
exp(-delta/T)

Solving a Timetabling Problem

- **Simulated Annealing**

```
def T_schedule(T):
    T = 0.999*T
    return T
```

		T						
		0.001	1	2	4	10	100	1000
Delta	0	100%	100%	100%	100%	100%	100%	100%
	1	0%	37%	61%	78%	90%	99%	100%
	2	0%	14%	37%	61%	82%	98%	100%
	4	0%	2%	14%	37%	67%	96%	100%
	10	0%	0%	1%	8%	37%	90%	99%
	100	0%	0%	0%	0%	0%	37%	90%
	1000	0%	0%	0%	0%	0%	0%	37%



Experiment HC and SA

```
Try the following initial solutions with Neighbourhood 1,
Neighbourhood 2 and Neighbourhood 3

def initial_solution():
    sol=np.random.randint(1,slots+1,disc)
    #sol=[1,1,1,1,1,1,1,1,1,1,1,1]
    #sol=[1,1,1,2,2,2,3,3,3,4,4,1]
    #sol=[1,1,4,2,2,2,3,3,3,4,4,1]
    #sol=[1,2,3,4,1,2,3,4,1,2,3,4]
    #sol=[3,3,3,4,4,2,1,1,1,2,2,4]
    return sol

hc = hillclimbing()
print(hc, ':', evaluate(hc))

sa = simulatedAnnealing()
print(sa, ':', evaluate(sa))
```

Solving a Timetabling Problem

- **Genetic Algorithms**
  - Chromossome: Solution sol (array with 12 numbers 1..4)
  - Initial Population: array with popSize of initial_solution()
  - Fitness: int evaluation(Solution sol) //-evaluation maximization
  - Crossover: TODO
  - Mutation: Solution neighbour1(Solution sol)
  - Selection: TODO (Roullete? Tournament?)

Solving a Timetabling Problem

e) Build a program that allows you to use the following methods to find the optimal (or sub-optimal) solution to this problem:

- Genetic Algorithms

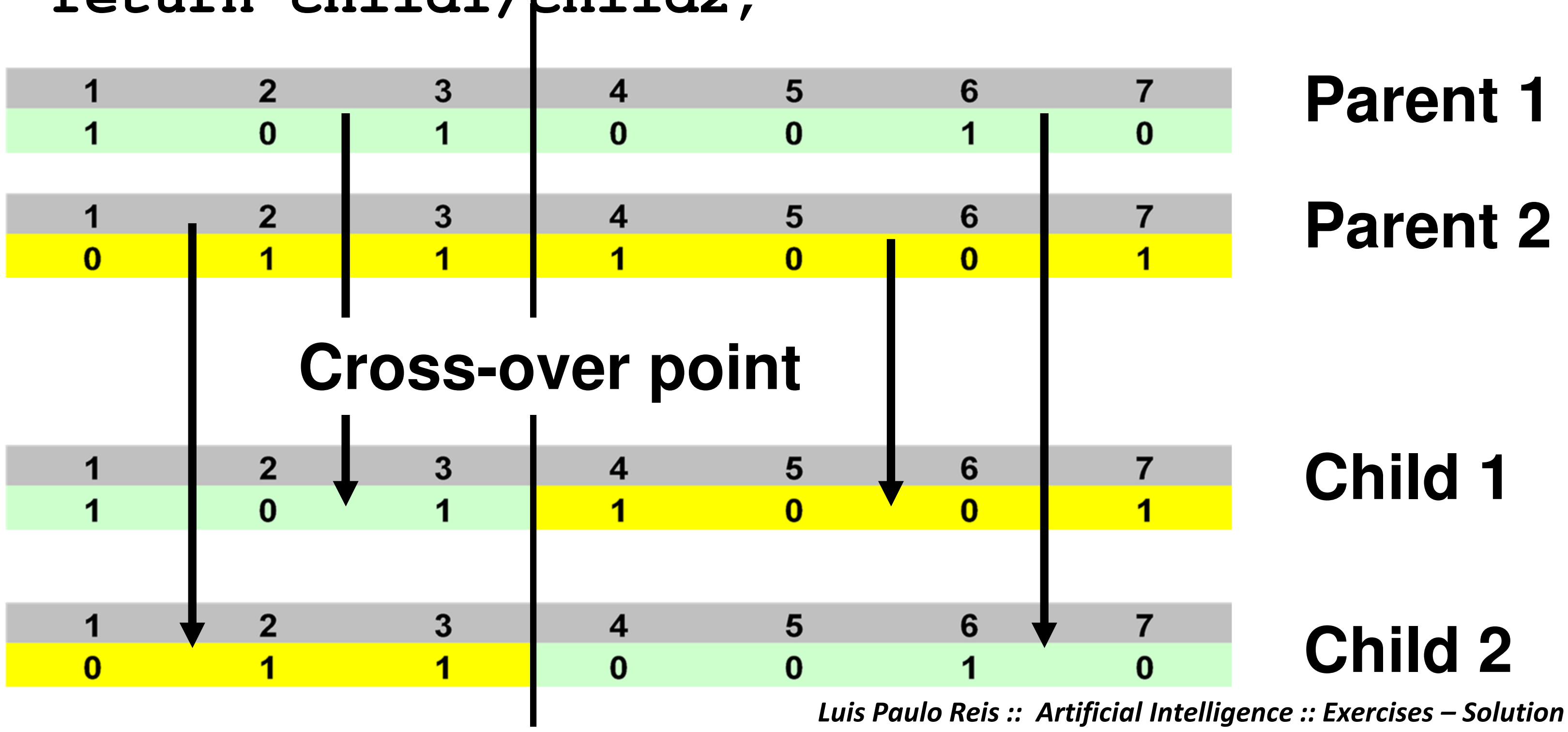
Note: Consider a stop_criteria of 50 iterations without improvement.

Code in C++:

```
Solution GeneticAlgorithms() {
  for(i=1; i<=PopSize; i++) sol[i] = initialSolution();
  it=0;
  do {
      evaluation;
      selection;
      crossover;
      mutation;
} while (it<50);
  return sol;
}
```

Crossover

```
Solution/Solution crossover(Solution par1, Solution
par2) {
      for(i=1; i<=Nd; i++)
            if(i<=Nd/2)
              child1[i]=par1[i]; child2[i]=par2[i];
            else
              child2[i]=par1[i]; child1[i]=par2[i];
      return child1/child2;
}
```



Tournament (Np=3)

```
Solution tournament(Solution par1, par2, par3) {
     //draw 3 different numbers 1..12
     if (evaluation(par1)>evaluation(par2) /\
           evaluation(par1)>evaluation(par3))
                 return par1;
     else if (evaluation(par2)>evaluation(par3))
           return par2;
           else return par3;
}
```

Tournament (Np=3)

```
Solution tournament() {
     p1=rand(1,popSize); p2=rand(1,popSize);
     p3=rand(1,popSize);  //numbers should be different
     if (evaluation(sol[p1])>evaluation(sol[p2]) /\
           evaluation(sol[p1])>evaluation(sol[p3])
                 return sol[p1];
     else if (evaluation(sol[p2])>evaluation(sol[p3])
           return sol[p2];
           else return sol[p3];
}
```

Solving a Timetabling Problem

e) Build a program that allows you to use the following methods to find the optimal (or sub-optimal) solution to this problem:

· Genetic Algorithms

Note: Consider a stop_criteria of 50 iterations without improvement.

```
Solution GeneticAlgorithms() {
  for(i=1; i<=PopSize; i++) sol[i] = initialSolution();
  it=0;
  do {
      for(i=1; i<=PopSize; i++)
            eval[i]=evaluation(sol[i]);
      selection;
      crossover;
      mutation;
} while (it<50);
  return sol;
}
```

IART - Artificial Intelligence

Exercise 3: Optimization/Meta-Heuristics

Luís Paulo Reis

LIACC – Artificial Intelligence and Computer Science Lab.
DEI/FEUP – Informatics Engineering Department, Faculty of Engineering of the University of Porto, Portugal
APPIA – Portuguese Association for Artificial Intelligence