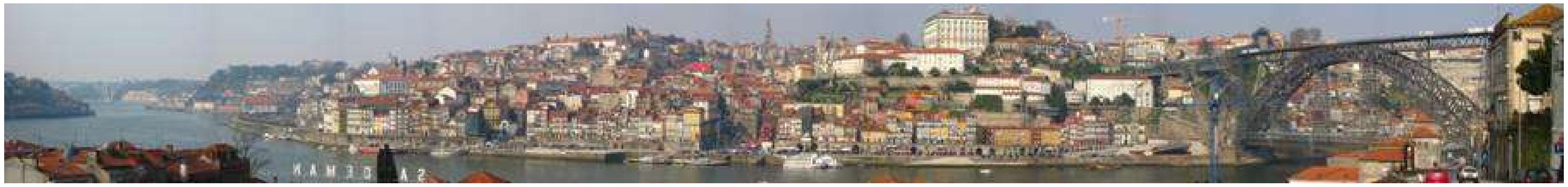
IART - Artificial Intelligence

Exercise 4: Optimization/Meta-Heuristics

Luís Paulo Reis, Henrique Lopes Cardoso

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



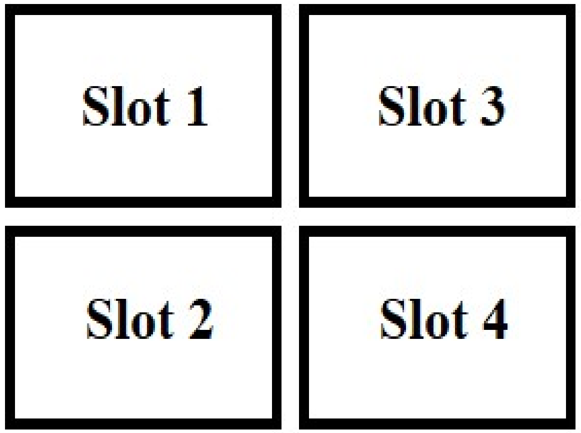
Exercise 4.1: Timetabling Problem Solving using Local Search/Simulated Annealing

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

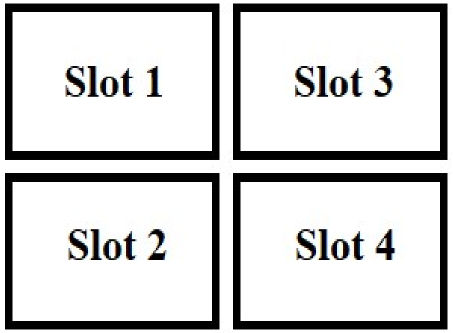
Data Representation

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.

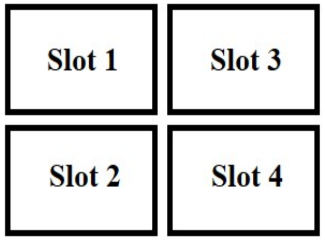
b) Build a function (on a language of your choice) that allows you to calculate an incompatibilities table, that is, for each pair of subjects (discipline) calculates the number of students who are enrolled in both.

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.

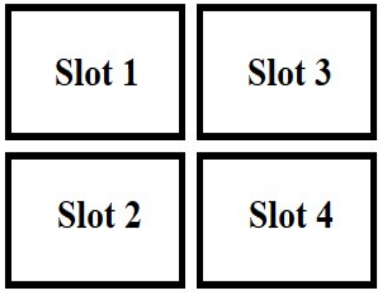
e) Build a program that allows you to use the following methods to find the optimal (or sub-optimal) solution to this problem:

- 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]). % What now? Another 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?**

```
Solution initialSolution() {
        …
}
```

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)

```
Solution initialSolution() {
        …
}
```

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)

```
Solution initialSolution() {
    for(i=1; i<=12; i++)
        sol[i]= rand(1,4);
    return sol;
}
```

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

```
int incompatib(int d1, int d2){
     …
     return nInc;
}
```

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 registrated at discipline 4

```
int incompatib(int d1, int d2){
     count = 0;
     for(a=1; a<=Na; a++)
          if(discStud[d1,a] /\ discStud[d2,a])
               count++;
     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.

```
int evaluate(Solution sol) {
  eval=0;
  for …
      …
  return eval;
}
```

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.

```
int evaluate(Solution sol) {
  eval=0;
  for (d1=1; d1<=Nd-1; d1++)
     for (d2=d1+1; d2<=Nd; d2++)
       if (sol[d1]==sol[d2])
           eval += incomp(d1,d2);
  return eval;
}
```

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], …

```
Solution neighbour1(Solution sol) {
  d1 = rand(1,Nd);
  …
  return sol;
}
```

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], …

```
Solution neighbour1(Solution sol) {
  d1 = rand(1,Nd);
  do
      new_slot = rand(1,Ns);
  while (new_slot == sol[d1]);
  sol[d1] = new_slot;
  return sol;
}
```

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], …

```
Solution neighbour2(Solution sol) {
  d1 = rand(1,Nd);
  do {
      d2=rand(1,Nd);
      } while (d1==d2 || sol[d1]==sol[d2]);
  aux=sol[d1]; sol[d1]=sol[d2]; sol[d2]=aux;
  return sol;
}
```

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], …

```
Solution neighbour3(Solution sol) {
      if (rand(0,1)==0) return neighbour1(sol);
      else return neighbour2(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.

```
Solution hillClimbing() {
  sol = …;
  do {
      …
  } until (stop_criteria(…));
  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.

```
Solution hillClimbing() {
  sol = initialSolution(); it=0;
  do {
      neig = neighbour3(sol); it++;
      if (eval(neig)<eval(sol)) //minimization
            { sol = neig; it=0; }
  } while (it<1000);
  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: $T_{i+1} = 0.9T_i$)

Note: Consider a stop_criteria of 1000 iterations without improvement.

```
Solution simulatedAnnealing() {
  sol = initialSolution();
  T = …
  do {
      T=…
      neig = neighbour(sol);
      …
  } until (stop_criteria());
  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.

```
Solution simulatedAnnealing() {
  sol = initialSolution(); it=0;
  T = Tinit(); // T = 1000;
  do {
      T=schedule(T);  // T = 0.9*T
      neig = neighbour(sol); it++;
      delta = eval(sol)-eval(neig);
      if (delta>0 || e^(-delta/T)>rand(0…1))
            { sol = neig; it=0; }
  } while (it<1000);
  return sol;
}
```

Solving a Timetabling Problem

- **Simulated Annealing**

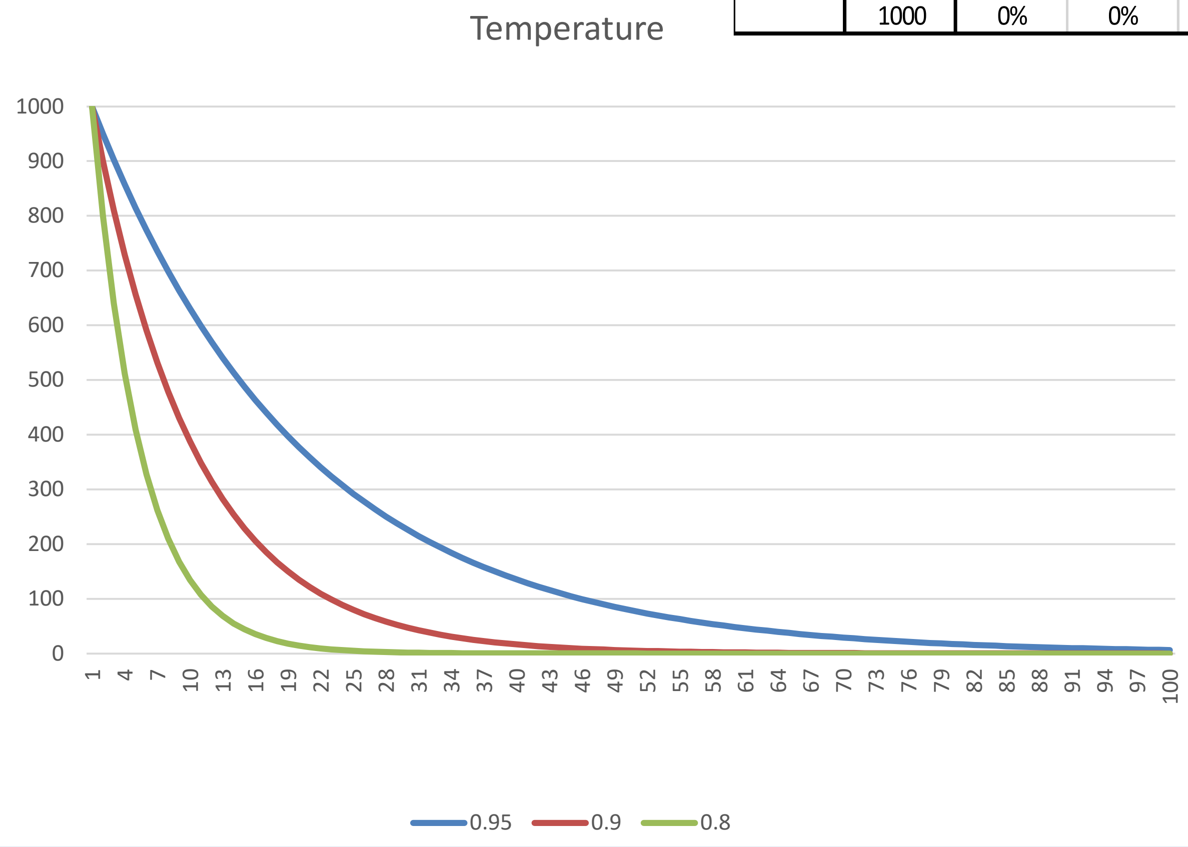
					T			
		0.001	1	2	4	10	100	1000
	0	100%	100%	100%	100%	100%	100%	100%
	1	0%	37%	61%	78%	90%	99%	100%
	2	0%	14%	37%	61%	82%	98%	100%
Delta	4	0%	2%	14%	37%	67%	96%	100%
	10	0%	0%	1%	8%	37%	90%	99%
	100	0%	0%	0%	0%	0%	37%	90%
	1000	0%	0%	0%	0%	0%	0%	37%

exp(-delta/T)

Solving a Timetabling Problem

- **Simulated Annealing**

					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%



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?)

U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

IART - Artificial Intelligence

Exercise 3: Optimization/Meta-Heuristics

Luís Paulo Reis, Henrique Lopes Cardoso

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