Artificial Intelligence I 2023/2024

Week 11 Tutorial and Additional Exercises

Simulated Annealing & Constraint Handling

School of Computer Science

29th April 2024

In this tutorial. . .

In this tutorial we will be covering

- Simulated Annealing.
- Constraint Handling.
- Examples of algorithm designs.

Exercise 1

Recall the algorithm for Simulated Annealing.

Algorithm 1: Simulated Annealing (maximisation).

```
Input: $T_0$: initial temperature, $T_f$: minimum temperature
Output: $\mathbf{x}$: final solution
$\mathbf{x} \leftarrow$ random initial solution;
$T \leftarrow T_0$;
repeat
    $\mathbf{x}' \leftarrow$ random neighbour of $\mathbf{x}$;
    if $quality(\mathbf{x}') \leq quality(\mathbf{x})$ then
        with probability $e^{\frac{\Delta E}{T}}$ : $\mathbf{x} \leftarrow \mathbf{x}'$
    else
        $\mathbf{x} \leftarrow \mathbf{x}'$
    $T \leftarrow schedule(T)$
until $T \leq T_f$ or $\mathbf{x}$ "stops changing";
return $\mathbf{x}$
```

where

- $\Delta E = quality(\mathbf{x}') - quality(\mathbf{x})$.
- $T > 0$ is the temperature.
- $schedule(T)$ is a non-increasing function that updates the temperature.
- $e \approx 2.71828\ldots$ is a constant.

How would you change algorithm 1 to do minimisation?

Exercise 1: Solution

One possible solution is the following:

Algorithm 2: Simulated Annealing (minimisation).

```
Input: $T_0$: initial temperature, $T_f$: minimum temperature
Output: $\mathbf{x}$: final solution
1  $\mathbf{x} \leftarrow$ random initial solution;
2  $T \leftarrow T_0$;
3  repeat
4      $\mathbf{x}' \leftarrow$ random neighbour of $\mathbf{x}$;
5      if $cost(\mathbf{x}') \geq cost(\mathbf{x})$ then
6          with probability $e^{\frac{\Delta E}{T}}$: $\mathbf{x} \leftarrow \mathbf{x}'$
7      else
8          $\mathbf{x} \leftarrow \mathbf{x}'$
9      $T \leftarrow schedule(T)$
10 until $T \leq T_f$ or $\mathbf{x}$ "stops changing";
11 return $\mathbf{x}$
```

where

- $\Delta E = cost(\mathbf{x}) - cost(\mathbf{x}')$.
- $T > 0$ is the temperature.
- $schedule(T)$ is a non-increasing function that updates the temperature.
- $e \approx 2.71828\ldots$ is a constant.

Exercise 2

We revisit the knapsack problem: Given N items with weights $w_1, \ldots, w_N$ and profits $p_1, \ldots, p_N$, find the set of items to be loaded into a knapsack so as to maximise the total profit of loaded items. The total weight of loaded items should be at most W. More formally:

$$\begin{array}{rl} \text{maximise} & f(\mathbf{v}) = \sum_{i=1}^{N} v_i p_i \\ \text{subject to} & g(\mathbf{v}) = \sum_{i=1}^{N} v_i w_i - W \leq 0 \end{array}$$

where $\mathbf{v} = (v_1, \ldots, v_N)$; and $\forall i \in \{1, \cdots, N\}$, v_i equals 1 if item i is loaded and 0 if not.

Exercise 2 (continued)

- We will apply the hill-climbing algorithm to find heuristic solutions to the knapsack problem.
- We start with some initial solution, then for each iteration, we generate neighbour solutions and pick the best of them.
- To use hill-climbing we need an initial solution and a method to generate neighbour solutions from a current solution.
- Exercise: Given the problem parameters, design two algorithms: One that will generate an initial solution and one that will generate neighbour solutions, given a current solution.
- You must ensure that any generated solution is feasible, so you might want to include a strategy to deal with the constraint!

Exercise 2: Solution

There are more than one possible solutions. Here is one:

Algorithm 3: Initial solution.

```
Input: $\mathbf{w} = (w_1, \dots, w_N)$: weights,
       $\mathbf{p} = (p_1, \dots, p_N)$: profits, $W$: maximum weight
Output: $\mathbf{v}$: initial solution
1 $\mathbf{v} \leftarrow (0, \dots, 0)$;
2 for $i = 1, \dots, N$ do
3     $r \leftarrow$ uniformly random value from $\{0, 1\}$;
4     if $r = 1$ then
5         If assigning $\mathbf{v}[i] \leftarrow 1$ would cause $W$ not to be exceeded, assign $\mathbf{v}[i] \leftarrow 1$, else assign $\mathbf{v}[i] \leftarrow 0$.
6     else
7         $\mathbf{v}[i] \leftarrow 0$.
8 return $\mathbf{v}$
```

Algorithm 4: Neighbourhood operator.

```
Input: $\mathbf{w} = (w_1, \dots, w_N)$: weights,
       $\mathbf{p} = (p_1, \dots, p_N)$: profits, $\mathbf{v}$: current solution, $W$: maximum weight
Output: $l_u$: list of neighbour solutions
1  $l_u \leftarrow []$;
2  for $i = 1, \dots, N$ do
3      $\mathbf{u} \leftarrow \mathbf{v}$;
4      $\mathbf{u}[i] \leftarrow 1 - \mathbf{v}[i]$;
5      $j \leftarrow 0$;
6      while $\mathbf{u}^T\mathbf{w} - W > 0$ do
7          $j \leftarrow j + 1$ (but skip the value of $i$);
8          $\mathbf{u}[j] \leftarrow 0$
9      if $\mathbf{u}^T\mathbf{w} - W \leq 0$ then
10         $l_u.add(\mathbf{u})$
11 return $l_u$
```

Exercise 2: Solution (continued)

- Algorithm 3 generates randomly the initial solution.
- Algorithm 4 generates the neighbours from a current solution at each iteration.
- We can then use hill-climbing or simulated annealing to pick the next neighbour from all candidates.
- Let us see an example using these algorithms.

Exercise 3

Consider the knapsack problem with $W = 10$ and parameters:

i	1	2	3	4	5
w_i	3	5	9	4	1
p_i	10	20	40	15	3

If the current solution is $\mathbf{v} = (0, 1, 0, 1, 0)$, find the next two neighbours using the same algorithm (repeated to the right). Write all neighbours we visit in the process, along with their weights and profits.

Algorithm 5: Neighbourhood operator (repeated).

```
Input: $\mathbf{w} = (w_1, \dots, w_N)$: weights,
       $\mathbf{p} = (p_1, \dots, p_N)$: profits, $W$:
       maximum weight, $\mathbf{v}$: current
       solution
Output: $l_u$: list of neighbour solutions
$l_u \leftarrow []$;
for $i = 1, \dots, N$ do
    $\mathbf{u} \leftarrow \mathbf{v}$;
    $\mathbf{u}[i] \leftarrow 1 - \mathbf{v}[i]$;
    $j \leftarrow 0$;
    while $\mathbf{u}^T\mathbf{w} - W > 0$ do
        $j \leftarrow j + 1$ (but skip the value
          of $i$);
        $\mathbf{u}[j] \leftarrow 0$
    if $\mathbf{u}^T\mathbf{w} - W \leq 0$ then
        $l_u.add(\mathbf{u})$
return $l_u$
```

Exercise 3: Solution

To find the first neighbour we search as follows:

$\boxed{W = 10}$

i	1	2	3	4	5
w_i	3	5	9	4	1
p_i	10	20	40	15	3

$$\begin{tikzcd}
 & & 01010^{w=9}_{p=35} \arrow[dll] \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr] & & \\
\underline{1}1010^{w=12}_{p=45} \arrow[d] & \boxed{0\underline{0}010^{w=4}_{p=15}} & 01\underline{1}10^{w=18}_{p=75} \arrow[d] & \boxed{010\underline{0}0^{w=5}_{p=20}} & \boxed{0101\underline{1}^{w=10}_{p=38}} \\
\boxed{1\underline{0}010^{w=7}_{p=25}} & & 0\underline{0}110^{w=13}_{p=55} \arrow[d] & & \\
 & & \boxed{001\underline{0}0^{w=9}_{p=40}} & &
\end{tikzcd}$$

The best neighbour is 00100 with profit 40 and weight 9.

Exercise 3: Solution (continued)

To find the second neighbour we search as follows:

$\boxed{W = 10}$

i	1	2	3	4	5
w_i	3	5	9	4	1
p_i	10	20	40	15	3

$$\begin{tikzcd}
& & 00100^{w=9}_{p=40} \arrow[dll] \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr] & & \\
\underline{1}0100^{w=12}_{p=50} \arrow[d] & 0\underline{1}100^{w=14}_{p=60} \arrow[d] & \boxed{00\underline{0}00^{w=0}_{p=0}} & 001\underline{1}0^{w=13}_{p=55} \arrow[d] & \boxed{0010\underline{1}^{w=10}_{p=43}} \\
\boxed{10\underline{0}00^{w=3}_{p=10}} & \boxed{01\underline{0}00^{w=5}_{p=20}} & & \boxed{00\underline{0}10^{w=4}_{p=15}} &
\end{tikzcd}$$

The best neighbour is 00101 with profit 43 and weight 10.

Exercise 3: Solution (continued)

What did we achieve with this algorithm?

- We went from a profit $35 \rightarrow 40 \rightarrow 43$ by searching 17 out of the 32 possible solutions.
- We can also use simulated annealing, which will have a chance to accept solutions with smaller profit than the current solution.
- This is just one of many possible methods to generate feasible neighbours from a current solution.

Exercise 3: Solution (continued)

Some things to try yourselves.

- Continue this algorithm to find the next best neighbour or conclude that there is none.
- Run the algorithm starting from a different initial solution. Write the best neighbour you find at each iteration, how the profit increases, and how many solutions were searched.
- Find the optimal solution to this problem with *brute-force search* by trying all 32 different solutions. Remember to compare only feasible solutions!
- The Hill Climbing operators that we designed here lead to some bias in that neighbours are more likely to have positions with zero in the beginning of the vector. Can you do better than that?

Exercise 4

Recall the knapsack problem formulation from Exercise 2. Design a strategy to deal with the constraint by modifying the objective function of this problem.

PS: Note that this is a maximisation problem, rather than a minimisation problem. So, you will need to change the general format of the strategy to deal with the constraint seen in the lecture.

Exercise 4: Solution

A possible strategy would be to modify the objective function to $f(v) - Q(v)$, where:

$Q(v) = v_g P g(v)^2$, P is a very large positive constant that should ideally cause any infeasible solution to have a worse quality than feasible ones, and $v_g = 1$ if the constraint $g(\mathbf{v})$ is violated and $v_g = 0$ otherwise.

PS: Note that we are subtracting rather than summing the penalty, because this is a maximisation problem.

Up next. . .

Advanced Material

Advanced Exercise 1

Implementing algorithms from the field of artificial intelligence has several similarities to implementing algorithms from other fields. However, one key difference is that several artificial intelligence algorithms involve probabilistic decisions. An example is Simulated Annealing, which makes use of the probability of replacing the current solution by a neighbour of equal or worse quality.

This question is intended as an example of how to make probabilistic decisions when implementing such algorithms. It is a challenge question, because you may never have implemented algorithms that involve probabilistic decisions before. However, once you understand how to make probabilistic decisions in the implementation, the same coding strategy can be used when implementing other algorithms that involve probabilistic decisions.

Advanced Exercise 1 (continued)

Consider that you would like to implement Simulated Annealing in Java. You have a pseudo random number generator able to give you pseudo random numbers in the interval $[0,1)$.[1]

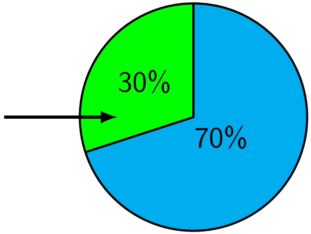
```
class Random:
public double nextDouble()
```

Write a piece of Java code that implements line 6 in algorithm 1.

[1]The notation $[0,1)$ means that 0 is included in the interval, but 1 is not.

Advanced Exercise 1: Solution

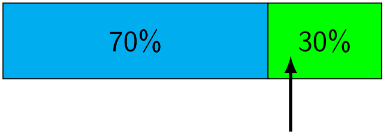
The probabilities of replacing or not replacing the current solution by a neighbour of equal or lower quality can be seen as proportions in a roulette wheel. For example, if the probability of replacing it is $e^{\frac{\Delta E}{T}} = 70\%$, then the probability of not replacing it would be $1 - e^{\frac{\Delta E}{T}} = 30\%$.



Exercise 5: Solution (continued)

The decision of whether or not to replace the current solution could be seen as spinning the wheel and checking whether the arm falls in the position corresponding to replacing or not replacing.

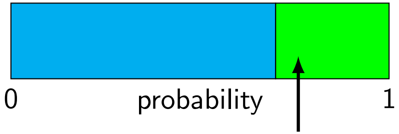
We could potentially see these proportions in a stripe as shown below:



Exercise 5: Solution (continued)

Therefore, picking a random position in the stripe could be seen as picking a random number in the interval [0,1). In the example above, if the number is in [0,0.7), then we fall into the 70% region corresponding to replacing the current solution. If the number is in [0.7,1), then it falls into the 30% region corresponding to not replacing the current solution.

If we consider a generic $probability = e^{\frac{\Delta E}{T}}$, then our stripe is as follows:



Exercise 5: Solution (continued)

The decision could thus be implemented as follows:

```
if(rand.nextDouble() < probability)
    currentSolution = randomNeighbour;
```

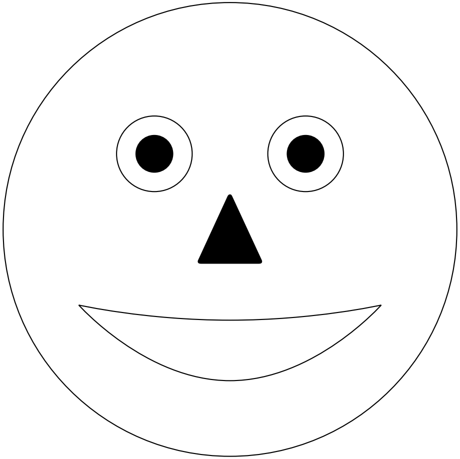
A piece of code to illustrate running these two lines for a given number of iterations and a pre-defined fixed probability value can be found here:
`https://canvas.bham.ac.uk/courses/65652/files/14332979?wrap=1`

The random seed used in this code is a parameter of the pseudo random number generator, as explained here:
`https://docs.oracle.com/javase/8/docs/api/java/util/Random.html`

Q&A

Any questions?

Until the next time. . .



Thank you for your attention!