Chalmers University of Technology

FFR135 - Artificial Neural Networks

Examples sheet 2

Jacopo Credi
(910216-T396)

September 24, 2015

Problem 1(a)

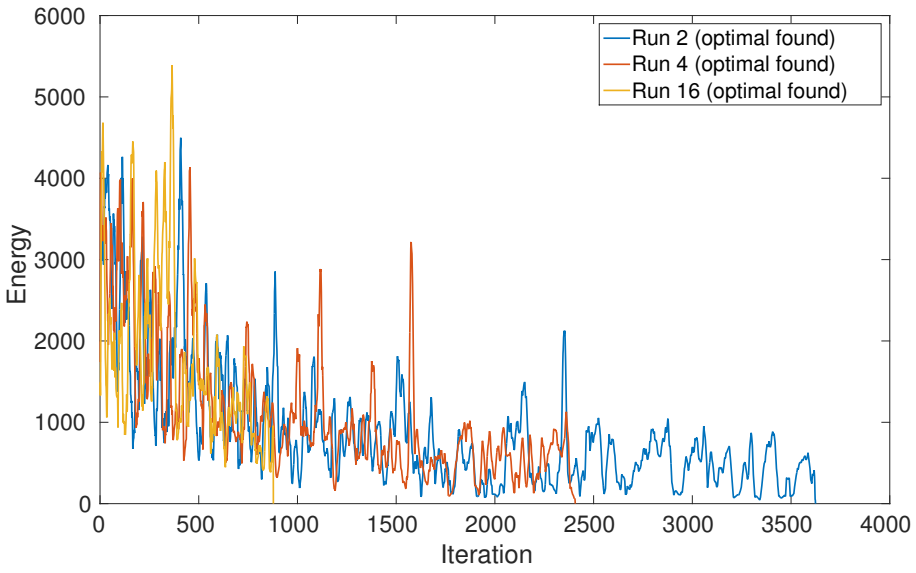

Figure 1: Evolution of energy over time for 3 runs randomly picked out of 100 runs with noise decrease rate $\alpha = 1 \cdot 10^{-6}$. Maximum allowed iterations $t_{\max} = 5 \cdot 10^4$. A moving average with a span of 20 points is displayed for clarity.

α	Optimal	$\langle t_{\text{conv}} \rangle$
$3 \cdot 10^{-4}$	32%	346
$1 \cdot 10^{-4}$	45%	415
$3 \cdot 10^{-5}$	46%	552
$1 \cdot 10^{-5}$	71%	985
$3 \cdot 10^{-6}$	90%	1914
$\mathbf{1 \cdot 10^{-6}}$	**97%**	**2810**
$3 \cdot 10^{-7}$	93%	3818
$1 \cdot 10^{-7}$	76%	3983
$3 \cdot 10^{-8}$	71%	4002
$1 \cdot 10^{-8}$	67%	4129

Table 1: Fraction of optimal solutions and average convergence time as a function of noise decrease rate α. Values were obtained by performing 100 runs for each value of α. Maximum allowed iterations $t_{\max} = 5 \cdot 10^4$.

A MATLAB implementation of the simulated-annealing algorithm was applied to `data_1`, with parameter β increasing linearly over time $\beta = \alpha \cdot t$ and candidate configuration being sampled in the neighbourhood of current configuration, as defined in Goldstein and Waterman (1987).

The energy evolution over time, plotted in Fig. 1 for three randomly chosen runs, reflects the expected behaviour for the SA algorithm. In the beginning, when the noise level is high, energy exhibits large fluctuations, which are then damped over time as the system is "cooled down". The average value of the energy decreases over time, since configurations (σ, μ) which causes the energy to decrease are more likely to be accepted.

The percentage of runs in which an optimal solution was found (i.e. the energy reaches zero) was observed to depend critically on α, and the same applies for the average number of iterations needed to reach zero energy, based on the runs in which an optimal solution was found (see Table 1).

The observed trends can be explained as follows: if α is too large, the system is cooled down too quickly and a large fraction of Markov Chains get stuck in a local minimum, but those chains which do converge to zero do so very quickly (e.g. $\langle t_{\text{conv}} \rangle = 552$ when $\alpha = 3 \cdot 10^{-5}$). On the other hand, if α is too low, the simulated annealing method has little advantage over a purely random search, and as α is decreased $\langle t_{\text{conv}} \rangle$ increases, and a larger and larger fraction of MC do not converge within the specified maximum number of iterations $t_{\max} = 5 \cdot 10^4$, even though they would probably converge if this value was larger.

With this value of $t_{\max}$, a noise decrease rate $\alpha = 1 \cdot 10^{-6}$ (in bold in Table 1) yields the largest percentage of optimal solutions and the lower runtime of the implemented program (as almost all chains converge to $H(S^{(k)}) = 0$ in a few thousands steps, well before $t_{\max}$ is reached).

Reference code

See MATLAB script `Main.m` and custom functions called in the script itself.

Problem 1(b)

Solution ID	Permutation σ	Permutation μ	Resulting permutation of $\mathbf{c}$	Energy
(1)	$(a_1\ a_6\ a_4\ a_2\ a_3\ a_5)$	$(b_1\ b_2\ b_6\ b_3\ b_5\ b_4)$	$(c_1\ c_{11}\ c_3\ c_7\ c_2\ c_6\ c_9\ c_4\ c_8\ c_5\ c_{10})$	0
(2)	$(a_1\ a_6\ a_4\ a_2\ a_3\ a_5)$	$(b_1\ b_2\ b_5\ b_3\ b_6\ b_4)$	$(c_1\ c_{11}\ c_3\ c_7\ c_2\ c_6\ c_8\ c_4\ c_9\ c_5\ c_{10})$	0
(3)	$(a_5\ a_3\ a_2\ a_4\ a_6\ a_1)$	$(b_4\ b_5\ b_3\ b_6\ b_2\ b_1)$	$(c_{10}\ c_5\ c_8\ c_4\ c_9\ c_6\ c_2\ c_7\ c_3\ c_{11}\ c_1)$	0

Table 2: Three optimal solutions found for `data_1`. The solution ID has no particular meaning and was just introduced for convenience. They were not chosen randomly, though, but in order to illustrate a point.

In general, the double-digest problem (DDP) has multiple optimal solutions. Three of the optimal solutions found by the implemented SA algorithm are shown in Table 2, but many more were found.

It was observed that solutions (2) and (3) in Table 2, in particular, can be reconstructed from solution (1). Permutation $\sigma^{(2)}$, in fact, is exactly the same as permutation $\sigma^{(1)}$, whereas permutation $\mu^{(2)}$ can be obtained from permutation $\mu^{(1)}$ by swapping elements b_5 and b_6. Why does such a swap lead to a different optimal solution? Because in solution (1) the sub-portion $(b_6\ b_3\ b_5)$ is completely "contained" in element a_3 (as illustrated in yellow in Fig. 2), therefore swapping any couple of these elements results in a zero-energy solution.

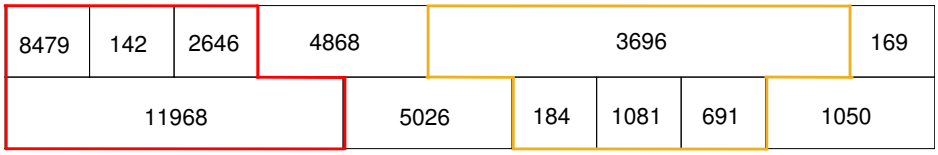

Figure 2: Graphical representation of solution (1) in Table 2. The drawing is not to scale.

We can indeed calculate the number of optimal solutions that can be constructed starting from solution (1) by *swapping* fragments in $\sigma^{(1)}$ contained in some $b_i \in \mu^{(1)}$ and viceversa. As made clear by Fig. 2, three fragments in $\sigma^{(1)}$, namely $a_1 = 8479$, $a_6 = 142$ and $a_4 = 2646$, are completely contained in $b_1 = 11968$ (see red box), and three fragments in $\mu^{(1)}$, namely $b_6 = 184$, $b_3 = 1081$ and $b_5 = 691$, are completely contained in $a_3 = 3696$ (see yellow box). Any couple of fragments within these triples can be swapped and the result will also be an optimal solution. The number of such allowed swaps is $3!\ 3! = 36$, thus there are at least 36 optimal solutions.

Moreover, any sequence obtained by *reflecting* an optimal solution is also an optimal solution. For instance, solution (3) in Table 2 is nothing but the reflection of solution (1). Therefore, we can conclude that the total number of optimal solutions for `data_1` is

$$N_{\text{opt}} = 2 \cdot 3! \cdot 3! = 72\ .$$

Since the configuration space for this problem is relatively small ($6!\ 6! = 518400$ configurations), a complete search was performed, verifying that indeed the number of optimal solution is 72.

Reference code

See MATLAB script `Main.m` and custom functions called in the script itself, in particular `CountSolutions.m` and `CompleteSearch.m`.

Problem 1(c)

A double-digest problem is completely specified by giving sequences

$$\begin{aligned} \boldsymbol{a} &= \{a_1, \dots, a_n\} \qquad \text{with} \quad a_1 \geq a_2 \geq \dots a_n \ , \\ \boldsymbol{b} &= \{b_1, \dots, b_m\} \qquad \text{with} \quad b_1 \geq b_2 \geq \dots b_m \ , \\ \boldsymbol{c} &= \{c_1, \dots, c_l\} \qquad \text{with} \quad c_1 \geq c_2 \geq \dots c_l \ . \end{aligned}$$

There are $n!/q!$ possible permutations of sequence $\boldsymbol{a}$, where q is the number of repeated elements in $\boldsymbol{a}$, and $m!/r!$ possible permutations of sequence $\boldsymbol{b}$, where r is the number of repeated elements in $\boldsymbol{b}$. The size of the configuration space for such a problem is therefore given by

$$N = \frac{n!\ m!}{q!\ r!} \ .$$

The probability of sampling the correct solution it out of N possible configuration if picking randomly is

$$p = \frac{1}{N} = \frac{q!\ r!}{n!\ m!} \ , \tag{1}$$

thus the probability of sampling the correct configuration after t unsuccessful samples is

$$P(\text{“success after } t \text{ trials”}) = (1-p)^t \ p = p \ e^{t \ \log(1-p)} \simeq p \ e^{-pt} \ , \tag{2}$$

where the last approximation comes from the Taylor expansion

$$\log(1-x) = -x + o(x) \quad \text{for} \quad x \to 0 \ .$$

In our case (`data_1`) we have $n = m = 6$ and $q = r = 0$, therefore $p = 1/518400 \simeq 1.929 \cdot 10^{-6}$ and the approximation can be considered sufficiently good.

Equation (2) allows us to approximate the average number of iterations needed to find the correct sequence under a random search as the expected value of a random variable $T_{\text{correct}} \sim \text{Exp}(p)$, i.e. exponentially distributed with parameter p:

$$\mathbb{E}(T_{\text{correct}}) \simeq {}^{1} \int_0^\infty t \ p \ e^{-p \ t} \ dt = \Big[-t \ e^{-p \ t} \Big]_0^\infty - \int_0^\infty e^{-p \ t} \ dt = \frac{1}{p} = \frac{n!\ m!}{q!\ r!} \ . \tag{3}$$

The expected number of iterations needed for a random search to find the correct solution is therefore equal to the size of the configuration space, i.e. for `data_1` we have

$$\mathbb{E}(T_{\text{correct}}) \simeq 518400 \ .$$

In comparison, the average number of iterations needed by the SA algorithm to find *an* optimal solution is 2 to 3 orders of magnitude lower (see Table 1). And even if the *optimal* solution found by the SA method might not be the *correct* one, once an optimal solution has been found, the set of other optimal solutions can be reconstructed by applying appropriate transformations.

[1] Here we are neglecting a correction factor of $e^p \sim 1.000001929$

Problem 2(a)

		Optimal configurations (zero energy) found for `data_2`
(1)	$\sigma^{(1)} =$	$(a_8\ a_6\ a_4\ a_{10}\ a_9\ a_1\ a_2\ a_5\ a_3\ a_7)$
	$\mu^{(1)} =$	$(b_8\ b_6\ b_7\ b_2\ b_1\ b_4\ b_{11}\ b_3\ b_9\ b_5\ b_{10})$
	Resulting $\mathbf{c}$ sequence =	$(c_{10}\ c_{16}\ c_8\ c_{18}\ c_{11}\ c_6\ c_{15}\ c_{14}\ c_4\ c_2\ c_5\ c_3\ c_7\ c_{20}\ c_{12}\ c_1\ c_{17}\ c_{13}\ c_9\ c_{19})$
	Iterations for convergence =	481441
(2)	$\sigma^{(2)} =$	$(a_7\ a_3\ a_5\ a_2\ a_1\ a_{10}\ a_9\ a_4\ a_6\ a_8)$
	$\mu^{(2)} =$	$(b_{10}\ b_5\ b_9\ b_3\ b_{11}\ b_4\ b_1\ b_2\ b_7\ b_6\ b_8)$
	Resulting $\mathbf{c}$ sequence =	$(c_{19}\ c_9\ c_{13}\ c_{17}\ c_1\ c_{12}\ c_{20}\ c_7\ c_3\ c_5\ c_2\ c_4\ c_{15}\ c_{14}\ c_6\ c_{11}\ c_{18}\ c_8\ c_{16}\ c_{10})$
	Iterations for convergence =	563669

Table 3: Optimal solutions found for `data_2` by the implemented SA algorithm with $\alpha = 10^{-7}$.

The simulated-annealing algorithm was successfully applied to solve the DDP specified in `data_2`, finding two optimal solutions (zero energy) displayed in Table 3.

In order to obtain these solutions, the algorithm was tested for several values of α, performing up to 10^6 iterations per run. The optimal solutions displayed above were then found by setting $\alpha = 10^{-7}$ and completing 50 runs, each with a maximum allowed number of iterations $t_{\max} = 7.5 \cdot 10^5$. Under these conditions, the percentage of successful runs was 4% and the average number of iterations needed for convergence to optimum was $\langle t_{\text{conv}} \rangle = 522555$.

As expected, the optimal solution is not unique. Again, we can notice that solution (2) can be constructed from solution (1), by reflecting it and then swapping fragments a_{10} and a_9. Two more optimal solutions can therefore be obtained by transforming solution (1): one by reflecting it without swapping fragments a_{10} and a_9, and another one by swapping fragments a_{10} and a_9 without reflection. This leads us to claim that there are 4 optimal solutions for this double-digest problem.

In comparison with `data_1`, this DDP seems to have fewer solutions, even though the number of fragments is larger. This is possible, albeit counterintuitive at first, because a larger number of fragments in sequence $\mathbf{c}$ means a larger number of constraints to satisfy in order for a solution to be optimal.

The difficulties in finding an optimal solution were expected, as the configuration space for this problem is much larger than the one in `data_1` (see **Problem 2b**).

Reference code

See MATLAB script `Main.m` and custom functions called in the script itself.

Problem 2(b)

In order to estimate the expected number of iterations needed to find the *correct* solution under a purely random search for DDP in `data_2`, we can use the approximation derived in part **1c** (Equation (3)). Since here $n \equiv| \boldsymbol{a} | = 10$, $m \equiv| \boldsymbol{b} | = 11$, and $q = r = 0$, we have

$$\mathbb{E}(T_{\text{correct}}) \simeq \frac{n!\, m!}{q!\, r!} = \frac{10!\, 11!}{1} = 144850083840000 \simeq 1.45 \cdot 10^{14} .$$

Clearly, the expected number of iterations needed to find the *correct* solution scales extremely quickly with the number of elements in sequences **a** and **b**. While in **Problem 1** it was even possible to perform a *complete* search over the entire configuration space, now this is unthinkable (on my laptop, at least).

The simulated-annealing algorithm, instead, did find a solution (even if it might not be the *correct* one) in about $5 \cdot 10^5$ iterations. This corresponds to a performance increase of 9 orders of magnitude, making the SA method a much better choice when dealing with this kind of combinatorial problems, even more so as the problem size increases.

Appendix: MATLAB code

Main.m

```
%MAIN
clear

% ================================================================== %
% Parameters
% ================================================================== %

% Use either data from data_1.rtf or data_2.rtf
a=[9979, 9348, 8022, 4020, 2693, 1892, 1714, 1371, 510, 451];
b=[9492, 8453, 7749, 7365, 2292, 2180, 1023, 959, 278, 124, 85];
c=[7042, 5608, 5464, 4371, 3884, 3121, 1901, 1768, 1590, 959, 899, 707,
   702, 510, 451, 412, 278, 124, 124, 85];

nRuns = 50;
maxTime = 750000; % use 50000 for data1, 750000 for data2
alpha = 1e-7; % 1e-6 for data1, 1e-7 for data2

% ================================================================== %

% Initialize output cell array
solutions = repmat(struct('a',a,'b',b,'c',c,'energy',1:maxTime,...
    'isOptimal',false,'runTime',nan), nRuns, 1 );

%% Main (parallel) loop
p =  TimedProgressBar( nRuns, 30, ...
    'Computing... Estimated time left: ', '   Completed: ', 'Completed
       in ' );
parfor run = 1:nRuns
    solutions(run) = DoubleDigestSimulatedAnnealing( a, b, c, maxTime,
       alpha);
    p.progress;
end
p.stop;
[ differentSolutions, nSolutions] = CountSolutions( solutions );

% Print some output to console
fprintf('\nFraction of optimal solutions: %f\n', ...
    mean(cell2mat(extractfield(solutions,'isOptimal'))));
fprintf('Average number of iterates for convergence to optimal: %i\n',
   ...
    int32(nanmean(extractfield(solutions,'runTime'))));
fprintf('Number of different optimal solutions found: %i\n', nSolutions)
   ;

```

```
40
41 %[f1, f2, f3, which] = PlotEnergy(solutions);
42
43 delete('timedProgressbar_Transient(deleteThis)_*.txt');
```

DoubleDigestSimulatedAnnealing.m

```
1  function solution = DoubleDigestSimulatedAnnealing(seqA, seqB, trueC,
       maxTime, alpha)
2  %DOUBLEDIGESTSIMULATEDANNEALING
3  %  solution = DoubleDigestSimulatedAnnealing(seqA, seqB, trueC, maxTime,
       alpha)
4  %
5  %  This function performs a simulated annealing search of an optimal
6  %  sequence solving the double digest problem.
7  %
8  %  INPUTS:
9  %   - sequence a (digested by A)
10 %   - sequence b (digested by B)
11 %   - sequence c (digested by both A and B)
12 %   - max allowed number of iterations
13 %   - increase rate for beta (it's also starting value for beta at t=1)
14 %
15 %  OUTPUT variable is a structure with the following fields:
16 %   - solution.a : best a-sequence found
17 %   - solution.b : best b-sequence found
18 %   - solution.c : best c-sequence found
19 %   - solution.energy : evolution of energy over time
20 %   - solution.isOptimal : logical value, true if the found sequence is
       optimal
21 %   - solution.runTime : number of iterations needed to find optimal (
       this value is NaN if optimal solution was not found)
22
23 %% Preliminary operations and initializations
24 nA = numel(seqA);
25 nB = numel(seqB);
26 energy = zeros(1,maxTime);
27 beta = alpha;
28 time = 1;
29 isOptimal = false;
30 runTime = nan;
31
32 % Initial guesses (random permutations)
33 guessA = seqA(randperm(nA));
34 guessB = seqB(randperm(nB));
35
```

```
% Initialize random logical 1d array (which to update between sigma and
    mu)
whichToUpdate = rand(maxTime+1,1) > 0.5;

% Corresponding c and initial energy
cTilde = DigestAB(guessA, guessB);
energy(time) = ComputeEnergy( trueC, cTilde);

%% Main loop
while time < maxTime

    time = time + 1;

    % Make new guesses for sequences a and b and corresponding sequence
       c
    if whichToUpdate(time) % update sigma
        newGuessA = SampleNeighbourPermutation(guessA);
        newGuessB = guessB;
    else % update mu
        newGuessA = guessA;
        newGuessB = SampleNeighbourPermutation(guessB);
    end
    cTilde = DigestAB(newGuessA, newGuessB);

    % Compute newEnergy and decide whether to accept new permutations
    newEnergy = ComputeEnergy( trueC, cTilde);

    if AcceptUpdate( beta, newEnergy, energy(time-1));
        guessA = newGuessA;
        guessB = newGuessB;
        energy(time) = newEnergy;
    else
        energy(time) = energy(time-1);
    end

    % Check whether an optimal configuration was found
    if newEnergy == 0
        isOptimal = true;
        runTime = time;
        energy(time+1:end) = [];
        break
    end

    % Update beta
    beta = alpha*time;
end

solution.a = guessA;
solution.b = guessB;
```

```
solution.c = cTilde;
solution.energy = energy;
solution.isOptimal = isOptimal;
solution.runTime = runTime;

end
```

SampleNeighbourPermutation.m

```
function newPermutation = SampleNeighbourPermutation( permutation )
%SAMPLENEIGHBOURPERMUTATION
%  newPermutation = SampleNeighbourPermutation( permutation ) returns an
%  output vector containing the same elements as the input vector, in
   the
%  same order except for 2 elements, which are swapped.

newPermutation = permutation;

toSwap = randperm(numel(permutation),2);

newPermutation(toSwap) = permutation(fliplr(toSwap));

end
```

DigestAB.m

```
function [ sortedC ] = DigestAB( sequenceA, sequenceB)
%DIGESTAB
%   [ sortedC ] = DigestAB( sequenceA, sequenceB, L ) returns the sorted
%   sequence c that would result from double digestion if sequenceA and
%   sequenceB were the correct fragment sequences. L is the sum of the
%   fragments' lengthts.

% Compute vectors of cumulative sums
cumulatedA = cumsum(sequenceA);
cumulatedB = cumsum(sequenceB);

% Concatenate cumsums and sort the resulting vector
sortedCumulatedC = union(cumulatedA, cumulatedB);

% Compute difference between adjacent elements
sequenceC = diff(sortedCumulatedC);

% Append first element of cumulative sum
sequenceC = [sortedCumulatedC(1) sequenceC];

```

```
% Sort resulting vector in descending order
sortedC = sort(sequenceC,'descend');

end
```

ComputeEnergy.m

```
function [ H ] = ComputeEnergy( c, cTilde )
%COMPUTEENERGY
%  H = ComputeEnergy( c, cTilde ) returns the energy function H(S^(k))
%  evaluated at the sequences c, cTilde

maxIndex = min(numel(c),numel(cTilde));
c = c(1:maxIndex);
cTilde = cTilde(1:maxIndex);

H = sum(((c-cTilde).^2)./c);

end
```

AcceptUpdate.m

```
function [ acceptUpdate ] = AcceptUpdate( beta, newH, oldH )
%ACCEPTUPDATE
%  acceptUpdate = AcceptUpdate( beta, newH, oldH ) returns the logical
%  value 1 (true) if the proposed update with energy newH is accepted, 0
%  (false) otherwise.

deltaH = newH - oldH;

% Compute acceptance probability
if deltaH > 0
    acceptanceProbability = exp(-beta*deltaH);
else
    acceptanceProbability = 1;
end

% Check whether the update is accepted or not
if rand < acceptanceProbability
    acceptUpdate = true;
else
    acceptUpdate = false;
end

end
```

CountSolutions

```
function [structDiffSolutions, nSolutions] = CountSolutions(outputData)
%COUNTSOLUTIONS

nRuns = length(outputData);
nA = numel(outputData(1).a);
nB = numel(outputData(1).b);

aSequences = extractfield(outputData, 'a');
aSequences = reshape(aSequences',[nA nRuns])';

bSequences = extractfield(outputData, 'b');
bSequences = reshape(bSequences',[nB nRuns])';

% Remove non-optimal solutions
isOptimal = cell2mat(extractfield(outputData,'isOptimal'))';
aSequences(~isOptimal,:) = [];
bSequences(~isOptimal,:) = [];

solutionsMatrix = [aSequences, bSequences];

differentSolutions = unique(solutionsMatrix,'rows');

nSolutions = size(differentSolutions, 1);

structDiffSolutions = repmat(struct('a', [] ,'b', []), nSolutions, 1);

for k = 1:nSolutions
    structDiffSolutions(k).a = differentSolutions(k,1:nA);
    structDiffSolutions(k).b = differentSolutions(k,nA+1:end);
end

end
```

PlotEnergy.m

```
function [f1, f2, f3, which] = PlotEnergy(solutions)
%PLOTENERGY

% randomly pick 3 chains
which = randi(length(solutions),[1 2]);
which = sort(which);

% if moving average
span = 5000;

```

```
smooth1 = smooth(solutions(which(1)).energy, span);
smooth2 = smooth(solutions(which(2)).energy, span);
smooth3 = smooth(solutions(which(3)).energy, span);

figure('units','normalized','outerposition',[0 0 1 1]);
f1 = plot(smooth1, 'LineWidth', 2);
hold on
f2 = plot(smooth2, 'LineWidth', 2);
f3 = plot(smooth3, 'LineWidth', 2);
hold off

% if all data points
% figure('units','normalized','outerposition',[0 0 1 1]);
% f1 = plot(solutions(which(1)).energy, 'LineWidth', 1.5);
% hold on
% f2 = plot(solutions(which(2)).energy, 'LineWidth', 1.5);
% f3 = plot(solutions(which(3)).energy, 'LineWidth', 1.5);
% hold off

set(gcf,'color','w');
set(gca,'fontsize', 28);

xlabel('Iteration');
ylabel('Energy');
pbaspect([1.618 1 1]);

for k = 1:3
    if solutions(which(k)).isOptimal
        converged{k} = ' (optimal found)';
    else
        converged{k} = ' (optimal not found)';
    end
end

str1 = ['Run ',num2str(which(1)), converged{1}];
str2 = ['Run ',num2str(which(2)), converged{2}];
str3 = ['Run ',num2str(which(3)), converged{3}];

legend([f1, f2, f3], str1, str2, str3);

end
```

CompleteSearch.m

```
function [nOptimalSolutions] = CompleteSearch( a, b, c )
%COMPLETESEARCH
%   Search all configuration space for optimal solutions of the DDP
```

```
%   specified by sequences a, b, c in input.
%   Returns number of optimal solutions.

% Save memory
if max([a, b]) < 65535
    a = uint16(a);
    b = uint16(b);
end

nOptimalSolutions = 0;

permA = perms(a);
permB = perms(b);

nA = size(permA,1);
nB = size(permB,1);

h = waitbar(0, 'Please wait...');
for aa = 1:nA

    tic
    currentA = permA(aa,:);

    parfor bb = 1:nB
        if isequal(c,DigestAB(currentA,permB(bb,:)));
            nOptimalSolutions = nOptimalSolutions + 1;
        end
    end
    waitbar(aa/nA, h);
    toc
end
close(h);

toc

end
```

TimedProgressBar.m

This is a nice function to print a waitbar on console when using parallelised for-loops. I found it on MatlabCentral/FileExchange. Credits: Antonio Jose Cacho.

URL: `https://www.mathworks.com/matlabcentral/fileexchange/46763-timedprogressbar`)

(Code is not reported here, see the above link.)