Linköping University

MAI0083

Heuristic Search Methodologies

Using a Genetic Algorithm to Solve the Rotating Workforce Scheduling Problem

Caroline Granfeldt

January 20, 2016

Contents

1 Introduction

The purpose of this project is to use a heuristic approach to the *rotating workforce scheduling problem*, further explained in Section 1.1. A method using a genetic algorithm, first developed by Mörz and Musliu in 2004 [1], is chosen for this. The algorithm is described in Section 2, while Section 3 presents the results. The results received by the algorithm is compared to those of the integer model developed in [2]. Section 4 contains a short discussion with some conclusions regarding the algorithm.

1.1 The Rotating Workforce Scheduling Problem

In many industries, it is required that work should be carried out 24 hours a day, 7 days a week. Typical work schedules in such contexts consist of a cycle repeating itself after a few weeks. The work is usually divided into different shifts, typically day (D), evening (E) and night (N) shifts. Figure 1 below shows an example of a rotating workforce schedule, where the empty spaces indicate the rest periods. Person A begins the cycle on week 1, while person B begins the cycle on week 2, person C begins the schedule on week 3, and so on. At the end of each week, each person moves down to the following week and the last person moves up to the first week. This means that the schedule repeats itself every six weeks.

Week	Mo	Tu	We	Th	Fr	Sa	Su
1			D	D	D	D	D
2	D	D				E	E
3	N	N	N	N			
4	E	E	E	E	E		
5	D	D	D	D	D		
6	D	D	D	D	D	D	D

Figure 1: *Example of a schedule with day, evening and night shifts.*

To be able to construct rotating workforce schedules, a workload matrix is required. Simply put, the matrix shows the staffing demand on each specific shift every week. Table 1 shows an example of a workload matrix, which matches the schedule given in Figure 1.

Table 1: *Example of a workload matrix.*

Shift	Mo	Tu	We	Th	Fr	Sa	Su
N	1	1	1	1			
D	3	3	3	3	3	2	2
E	1	1	1	1	1	1	1

2 Genetic Algorithm

A genetic algorithm is a search heuristics that mimics the process of natural selection by using techniques inspired by evolution like inheritance, mutation, crossover and selection. Algorithm 1 gives an overview of the algorithm used in this project.

Algorithm 1 The genetic algorithm used in this project

1: Initialize a population of N individuals
Repeat
2: Generate offsprings from individuals in the population (parents)
3: Mutate the offsprings
4: Evaluate the parents and the offsprings
5: Replace the population with individuals from the parents and the offsprings according to their evaluated fitness
Until An individual has good enough fitness
Output Best individual found

To begin with, a population of N random individuals is initialized. These individuals generate offsprings according to the reproduction phase explained in Section 2.3. Both the parents and children are then evaluated according to the fitness calculations in Section 2.4.1, and given different probabilities of survival until the next generation. This is repeated until some stopping criteria is satisfied. In this implementation, that stopping criteria is that the fitness of any individual should be zero. Then, that individual is given as the output.

2.1 Representation of Individuals

An individual of the population is a full cyclic schedule. It is represented by an $w \times d$-matrix S, where each element $s_{wd} \in A$ corresponds to a shift or day off at week w, day d.

2.2 Generation of Initial Population

The initial population is generated randomly, but in a way such that the staffing demand is always satisfied. This is done by creating each schedule a day at a time. For every day, required work shifts are randomly distributed among the weeks/employers. Thus, the staffing demand will always be exactly fulfilled for every created schedule/individual. As we will see in Sections 2.3.1 and 2.3.2, the crossover and mutation operators are designed so that this continues to hold for every generation.

2.3 Reproduction

After an initial population is created, it is necessary to have methods for reproduction so that new generations can be produced. This section explains how

this is implemented in the algorithm.

2.3.1 Crossover

The crossover operator (also known as recombination) uses two random solutions, known as parent solutions, from the population. These parents breed with a probability, chosen to be 0.5 in the algorithm, and produces two children solutions.

The operator uses random crossover, where columns are swapped between the individuals. The algorithm can be broken down into the following steps:

1. Choose a random number $x, x \in 1, ..., w - 1$
2. For x times, choose a unique random column c_x
3. Swap the chosen columns between the individuals

First, it is randomly decided how many column swaps that should be made. Then, the columns to be swapped are chosen randomly. Observe that a column can never be chosen twice or more since this would affect the number of swaps and consequently lowering the value of x. The two new solutions received by these operations are the children.

2.3.2 Mutation

To increase the genetic diversity, mutation is used. After a crossover, mutation appears in a child with a certain probability. This probability is set to be very high (0.8) in this algorithm, as Mörz and Musliu [1] stated that it is a powerful operator.

The move operator consists in swapping two elements in an individual. To ensure that the staffing demands are still fulfilled, the swap only occurs within a column. The column, as well as the two elements, are chosen randomly.

2.4 Selection

In the selection phase, N individuals from the parents and children are chosen to form the next generation. This is done by evaluating the fitness of all individuals and assign them probabilities to be chosen thereafter.

The fitness of an individual is simply the sum of all constraints it violates. Hence, lower values are preferred. The exact calculations are explained in Section 2.4.1. However, as mentioned above, the algorithm stops when an individual has the fitness value zero. Thus, all the constraints are satisfied. Consequently, this means that we only work with hard constraints in this genetic algorithm.

After the fitness has been calculated, the individuals are subdivided into classes depending on their fitness. This yields a total of p classes, where p is any number between 1 and the total number of individuals (parents + children). Class $x, x \in 1, ..., p$, is given the probability 0.5^x to be chosen (note that this assignment favours individuals with better fitness). If a class is chosen, the first

individual in that class is added to the new population and then removed from the pool of individuals. If this leads to an empty class, then the entire class is removed from the pool.

Observe that the assigned class probabilities do not sum up to exactly 1 (especially after a few iterations when classes have been removed). Thus, it is possible to choose a "void" class. If this void class is selected, the algorithm simply redraws until an existing class is picked. (Imagine the roulette wheel selection, where each pie slice corresponds to a class. With the above implementation, we receive "eaten" slices. If one of these slices are chosen by the spinning pointer, then the pointer is simply re-spun.)

2.4.1 Fitness Evaluation

To calculate the fitness, a number of parameters need to be defined. Table 2 shows these.

Table 2: *The different parameters needed for the fitness calculation.*

Notation	Description
S	The schedule (individual) to calculate the fitness for
m	The number of different shift types, where the m-th type is the day-off shift
NW_s	The number of work blocks in schedule s
ND_s	The number of days-off blocks in schedule s
NS_{js}	The number of shift sequences of shift type j in schedule s
WB_{is}	A work block i in schedule s
DOB_{is}	A days-off block i in schedule s
SB_{ijs}	The i-th shifts' block of shift type j in schedule s
$MAXW$	Maximum permitted length of blocks of work days
$MINW$	Minimum permitted length of blocks of work days
$MAXS_j$	Maximum permitted length of periods of consecutive shifts of type j
$MINS_j$	Minimum permitted length of periods of consecutive shifts of type j
C	A shift change matrix of dimension $m \times m \times m \times m$. Element c_{ijkl} is 1 if a sequence of shifts (i, j, k, l) is permitted, otherwise 0.

The fitness is then calculated as follows:

$$f = f_1 + f_2 + f_3 + f_4 \tag{1}$$

where

$$f_1 = \sum_{i=1}^{NW_s} Distance(WB_{is}, [MINW, MAXW]) \tag{2}$$

$$f_2 = \sum_{i=1}^{ND_s} Distance(DOB_{is}, [MINS_m, MAXS_m]) \tag{3}$$

$$f_3 = \sum_{j=1}^{m-1} \sum_{i=1}^{NS_{js}} Distance(SB_{ijs}, [MINS_j, MAXS_j]) \tag{4}$$

$$f_4 = NumOfNotAllowedShiftSeq(S, C) \tag{5}$$

The calculations contains two functions: *Distance(XBlock,range)* and *NumOfNotAllowedShiftSeq(S)*.

The function *Distance(XBlock,range)* returns 0 if the length of *XBlock* is within the allowed *range* of two numbers. Otherwise, it returns the distance from that range. For example, if the feasible range is 3-7 and the length of the block is 2, then the function will return 1.

The function *NumOfNotAllowedShiftSeq(S,C)* calculates, by using the shift change matrix C, how many violations of allowed shift sequences the schedule S has.

Thus, to summarize, the following constraints are handled in this algorithm:

- Minimum and maximum length of consecutive work days
- Minimum and maximum length of consecutive days off
- Minimum and maximum length of consecutive work days of a specific shift type (for example, it might be infeasible to work more than 3 night in a row, but neither day or evening shifts have this restriction)
- Specific sequences of shifts are prohibited, and this is currently the only constraint that is corrigible in different problem instances by using a different C-matrix. For example, if it is infeasible to work a day shift directly after a night shift then the sequence to forbid is (n, d, x, x), where n is a night shift, d is a day shift and x is any other shift.

3 Results

The algorithm is implemented in Matlab (the code can be found in Appendix A). Section 3.1 shows the different shifts used, the staffing demand and the implemented constraints. The resulting schedule is given in Section 3.2, where the results from the algorithm are compared to the results given by the integer model, with the same conditions and constraints, designed in [2].

3.1 Shifts, Staffing Demand and Constraints

The different shifts used are seen below in Table 3, while Table 4 shows the workload matrix.

Table 3: *The different shifts.*

Shift	Time
N	$22^{00} - 06^{00}$
D	$06^{00} - 14^{00}$
E	$14^{00} - 22^{00}$

Table 4: *The workload matrix.*

Shift	Mo	Tu	We	Th	Fr	Sa	Su
N	1	1	1	1			
D	3	3	3	3	3	3	3
E	1	1	1	1	1		

The constraints used are the following:

- It is prohibited to work more than 6 consecutive days, regardless of shift type.
- After the last night shift in a work block, there should be at least 50 hours off until the next work shift starts.

3.2 Resulting Schedules

The schedule given by the genetic algorithm is seen in Figure 2a, while Figure 2b is given by the integer model.

Week	Mo	Tu	We	Th	Fr	Sa	Su
1	D	D	D	D	D		D
2	D		E	D		D	D
3	E	E		E	D	D	
4	D	D	D		D	D	
5	N	N	N	N			
6		D	D	D	E		D

(a) *The genetic algorithm solution*
Solution time: 4.65 s

Week	Mo	Tu	We	Th	Fr	Sa	Su
1	D	D	D	D	D		D
2	D	E	E	E	E		D
3	N	N	N	N			
4	E		D	D	D	D	D
5		D				D	
6	D	D	D	D	D	D	

(b) *The integer model solution*
Solution time: 7.19 s

Figure 2: *The resulting schedules.*

Both solutions are feasible with regard to the constraints. At a first glance, it looks as though the genetic algorithm performs better with regard to the solution times. However, the solution time for the genetic algorithm varies on different runs. This is because the genetic algorithm is not deterministic. As described in Section 2, it contains several steps where randomness is involved (for instance the generation of an initial population, or the crossover operator). Table 6 shows some interesting data received during the 50 runs. As is clear by the table, the variance on the solution times and number of generations is quite large.

Table 6: *Data received after 50 runs of the genetic algorithm.*

Condition	Result
Shortest solution time	0.27 s
Longest solution time	149.48 s
Average solution time	16.41 s
Minimum number of generations	37
Maximum number of generations	21958
Average number of generations	2343

4 Discussion

The method is very limited in its present form since it is difficult to implement constraints simply by forbidding shift sequences. Currently, it is not possible to have any constraints which need longer time span than 4 days. (An example of such a constraint is that is should be infeasible to work two weekends in a row.) However, this problem can of course be solved by adding a fitness function which looks at precisely weekend distribution. The disadvantage though is that the algorithm is no longer general if this is done.

Another disadvantage of this algorithm compared to the integer model in [2] is that it can not handle soft constraints. The idea with soft constraints is that you can penalize certain attributes and sort constraints in a hierarchical order. This is difficult to implement in this algorithm since we always look for an individual with a fitness value of zero. An idea is to use two different fitness function values, where one represents the hard constraints and the other represents the soft constraints. The problem with this is that it is hard to formulate stopping criteria (unless both fitness function values are zero) other than a preset number of iterations.

Furthermore, the genetic algorithm in this project is a heuristic. Thus, it is not guaranteed that an optimal solution can always be found. The integer model solves this by using soft constraints, but, as stated above, it is difficult to use soft constraints in this algorithm. Moreover, the results show that the integer model outperforms the genetic algorithm in term of solution times. (This comparison is not very scientific though since runs where done on computers with different performance, and 50 runs is far too few to make a judgement.) In conclusion, this genetic algorithm may not be worthwhile to develop further since better alternatives exist.

5 References

[1] N. Musliu and M. Mörz. Genetic Algorithm for Rotating Workforce Scheduling Problem. In *Second IEEE International Conference on Computational Cybernetics*, 2004.

[2] C. Granfeldt. Rotating Workforce Scheduling. Master's thesis, Linköping University, 2015.

A Appendix

```
% Main Loop

%-------------------------------------------------------------------------
% Setup problem data
%-------------------------------------------------------------------------
setupFile;

%-------------------------------------------------------------------------
% Genetic Algorithm
%-------------------------------------------------------------------------
nrRuns = 50;
time = zeros(1,nrRuns);
generations = zeros(1,nrRuns);

for k=1:nrRuns
tic;

% Define size of population
N = 2*n;

% Generation of initial population
P_initial = GenerateInitialPop(R,A,n,w,N);

% Generate new population
P = P_initial;

gen = 1; % generation
while(1)
    Children = zeros(size(P));

    % Crossover
    crossoverProb = 0.5;
    nrChildren = 0;
    for i=1:N/2
        p1 = randi([1,12],1);
        while(1)
            p2 = randi([1,12],1);
            if p2 ~= p1
                break;
            end
        end
        if rand < crossoverProb
            [child1,child2] = CrossoverOperator(P(:,:,p1),P(:,:,p2),w);
            nrChildren = nrChildren+1;
```

```
            Children(:,:,nrChildren) = child1;
            nrChildren = nrChildren+1;
            Children(:,:,nrChildren) = child2;
        end
    end

    Children = Children(:,:,1:nrChildren);

    % Mutation of children
    mutationProb = 0.8;
    for i=1:nrChildren
        if rand < mutationProb
           S = Children(:,:,i);
           S_mutated = MutationOperator(S,n,w);
           Children(:,:,i) = S_mutated;
        end
    end

    % Calculate fitness of parents and children
    fitness = zeros(N+nrChildren,2);
    fitness(:,1) = 1:N+nrChildren;

    for i=1:N
        f = CalculateFitness(P(:,:,i),MAXW,MINW,MINS,MAXS,m,C);
        fitness(i,2) = f;
    end
    for i=1:nrChildren
        f = CalculateFitness(Children(:,:,i),MAXW,MINW,MINS,MAXS,m,C);
        fitness(N+i,2) = f;
    end

    feasibleSolution = find(fitness(:,2) == 0);
    if ~isempty(feasibleSolution)
        sol = feasibleSolution(1);
        if sol <= N
            %disp(P(:,:,sol));
            f = CalculateFitness(P(:,:,sol),MAXW,MINW,MINS,MAXS,m,C);
        else
            %disp(Children(:,:,sol-N));
            f = CalculateFitness(Children(:,:,sol-N),MAXW,MINW,MINS,MAXS,m,C);
        end
        generations(k) = gen;
        break;
    end

    Classification = ClassifyIndividuals(fitness);

    P_new = SelectionOperator(P,Children,Classification);
    P = P_new;
    gen = gen+1;
end

time(k) = toc;
end
%-------------------------------------------------------------------------
```

```
% setupFile
%--------------------------------------------------------------------------

% Number of employees
n = 6;

% Length of schedule
w = 7;

% Set of m shifts, where shift m = day off
A = [1 2 3 0];
m = length(A);
%A = ['n' 'd' 'e' 'x'];

% The workforce matrix
R = [1 1 1 1 0 0 0;
     3 3 3 3 3 3 3;
     1 1 1 1 1 0 0];

%--------------------------------------------------------------------------
% Constraints
%--------------------------------------------------------------------------

% Sequences of shifts permitted to be assigned to employees
C = ones(m,m,m,m);

% Must have 50 hours off after night shift
for i=1:m
    for j=1:m
        C(1,2,i,j) = 0;
        C(1,3,i,j) = 0;
    end
    for j=1:m-1
        C(1,m,j,i) = 0;
    end
end
C(1,m,m,1) = 0;
C(1,m,m,2) = 0;

% Maximum and minimum length of periods of consecutive shifts
MAXS = [6 6 6 30];
MINS = [1 1 1 1];

% Maximum and minimum length of blocks of workdays
MAXW = 6;
MINW = 1;
```

```
% GenerateInitialPop
function [P] = GenerateInitialPop(R,A,n,w,N)

% Create individuals for the population
P = zeros(n,w,N);
for i=1:N
    S = GenerateSchedule(R,A,n,w);
    P(:,:,i) = S;
end

end


function [S] = GenerateSchedule(R,A,n,w)

S = zeros(n,w);
for i=1:w
    v = zeros(n,1); % column in schedule

    shiftArray = [];
    shift = 1;
    for r=R(:,i)'
        temp = A(shift)*ones(1,r);
        shiftArray = [shiftArray,temp];
        shift = shift+1;
    end
    availEmployees = 1:n;
    for j=shiftArray
        r = randi([1,length(availEmployees)],1); % get random employee
        e = availEmployees(r);
        v(e) = j;
        availEmployees(r) = [];
    end

    S(:,i) = v;

end

end
```

```
% CrossoverOperator
% Swap x random columns
function [S3,S4] = CrossoverOperator(S1,S2,w)

S3 = S1;
S4 = S2;

x = randi([1,w-1],1);
C = zeros(1,x);
for i=1:x
    while(1)
        c = randi([1,w],1);
        if(isempty(find(C == c, 1)))
            break;
        end
    end
    C(i) = c;

    S3(:,c) = S2(:,c);
    S4(:,c) = S1(:,c);
end

end
```

```
% MutationOperator
% Performs a mutation
function [S_new] = MutationOperator(S,n,w)

element1 = randi([1,n],1);
while(1)
    element2 = randi([1,n],1);
    if element1 ~= element2
        break;
    end
end
column = randi([1,w],1);

S_new = S;

S_new(element1,column) = S(element2,column);
S_new(element2,column) = S(element1,column);

end
```

```
% CalculateFitness
% Calculates the fitness of an individual
function [fitness] = CalculateFitness(S,MAXW,MINW,MINS,MAXS,m,C)

% NW = number of work blocks
% WB[i] = a work block i
[WB,NW] = FindWorkBlocks(S);
f1 = 0;
for i=1:NW
    f1 = f1 + CalculateDistance(WB(i,:),[MINW,MAXW]);
end

% ND = number of days of blocks
% DOB[i] = a days off block i
[DOB,ND] = FindDaysOffBlocks(S);
f2 = 0;
for i=1:ND
    f2 = f2 + CalculateDistance(DOB(i,:),[MINS(m),MAXS(m)]);
end

% NS[j] = number of shift sequences blocks of shift j
% SB[i,j] = the i-th shifts block of shift j
[SB,NS] = FindShiftSequencesBlocks(WB,m);
f3 = 0;
for j=1:m-1
    for i=1:NS
        f3 = f3 + CalculateDistance2(SB(i,j),[MINS(j),MAXS(j)]);
    end
end

%f4 = 0;
f4 = ForbiddenShiftSequences(S,C,m);

fitness = f1 + f2 + f3 + f4;

end

function [WB,NW] = FindWorkBlocks(S)
[n,w] = size(S);
WB = zeros(n*w,n*w);

row = 1;
column = 1;
for i = 1:n
    for j=1:w
        s = S(i,j);

        if s == 0
            row = row+1;
            column = 1;
        else
            WB(row,column) = s;
            column = column+1;
        end
```

```
    end
end

% Check if last week has a work block overlapping the first week
WB = WB(any(WB,2),:);
[NW,~] = size(WB);
if (and(S(n,w)~=0,S(1,1)~=0))
    a = WB(1,:);
    a(a==0) = [];
    b = WB(NW,:);
    b(b==0) = [];
    c = [b,a,zeros(1,n*w-length(b)-length(a))];
    WB(NW,:) = c;
    WB(1,:) = [];
    NW = NW-1;
end
WB = WB(:,any(WB));

end

function [DOB,ND] = FindDaysOffBlocks(S)

[n,w] = size(S);
DOB = zeros(n*w,n*w);

row = 1;
column = 1;
for i = 1:n
    for j=1:w
        s = S(i,j);

        if s ~= 0
            row = row+1;
            column = 1;
        else
            DOB(row,column) = 1;
            column = column+1;
        end

    end
end

% Check if last week has a days off block overlapping the first week
DOB = DOB(any(DOB,2),:);
[ND,~] = size(DOB);
if (and(S(n,w)==0,S(1,1)==0))
    a = DOB(1,:);
    a(a==0) = [];
    b = DOB(ND,:);
    b(b==0) = [];
    c = [b,a,zeros(1,n*w-length(b)-length(a))];
    DOB(ND,:) = c;
    DOB(1,:) = [];
    ND = ND-1;
end
```

```
DOB = DOB(:,any(DOB));

end


function [SB,NS] = FindShiftSequencesBlocks(WB_in,m)
[NW,len] = size(WB_in);
WB = [WB_in , zeros(NW,1)];
SB = zeros(len,m);

for i=1:NW
    s_prev = WB(i,1);
    counter = 0;
    for j=1:len+1
        s = WB(i,j);
        if s == s_prev
            counter = counter+1;
        else
            SB(counter,s_prev) = SB(counter,s_prev)+1;
            if s == 0
                break;
            end
            counter = 1;
            s_prev = s;
        end
    end

end

NS = len;

end

function [nr] = ForbiddenShiftSequences(S,C,m)
nr = 0;

[n,w] = size(S);
S_string = zeros(1,n*w);
index = 1;
for i=1:n
    S_string(index:index+w-1) = S(i,:);
    index = index+w;
end

for i=1:n*w
    if i+1 > n*w
        j = i-n*w;
    else
        j=i;
    end
    if i+2 > n*w
        k = i-n*w;
    else
        k=i;
    end
```

```
    if i+3 > n*w
        l = i-n*w;
    else
        l=i;
    end
    a = S_string(i);
    b = S_string(j+1);
    c = S_string(k+2);
    d = S_string(l+3);
    % Ugly workaround index 0
    if a == 0
        a = m;
    end
    if b == 0
        b = m;
    end
    if c == 0
        c = m;
    end
    if d == 0
        d = m;
    end

    if C(a,b,c,d) == 0
        nr = nr+1;
    end
end

end

function [d] = CalculateDistance(XBlock,range)

% remove empty columns
XBlock = XBlock(any(XBlock),:);

x = length(XBlock);

if x < range(1)
    d = abs(range(1)-x);
elseif x > range(2)
    d = abs(x-range(2));
else
    d = 0;
end

end



function [d] = CalculateDistance2(x,range)

if x == 0
    d = 0;
    return;
end
```

```
if x < range(1)
    d = abs(range(1)-x);
elseif x > range(2)
    d = abs(x-range(2));
else
    d = 0;
end

end
```

```
% Classify Individuals
function [Class] = ClassifyIndividuals(fitness)
[N,~] = size(fitness);
Class = [fitness,zeros(N,2)];

[~,d2] = sort(Class(:,2));
Class = Class(d2,:);
f_prev = 0;
c = 1;
p = 0;
for i=1:N
    f = Class(i,2);
    if f ~= f_prev
        p = p+(1/2)^c;
        c = c+1;
    end
    Class(i,3) = c-1;
    f_prev = f;

    Class(i,4) = p;
end

end
```

```
% SelectionOperator
function [Pop] = SelectionOperator(Parents,Children,Class)

[~,~,nrParents] = size(Parents);
[~,~,nrChildren] = size(Children);
Pop = zeros(size(Parents));
[nrInd,~] = size(Class);
% Get table of intervals
nrClasses = max(Class(:,3));
ClassTable = zeros(nrClasses,3);
ClassTable(:,1)=1:nrClasses;
```

```
p = 0;
for i=1:nrClasses-1
    p = p + (1/2)^i;
    ClassTable(i,3) = p;
    ClassTable(i+1,2) = p;

end
p = p + (1/2)^nrClasses;
ClassTable(nrClasses,3) = p;

classesLeft = [1:nrClasses];
indLeft = [1:nrInd];
indsAdded = 0;
while(1)
    p = rand;

    class = 0;
    for i=1:nrClasses
        if p < ClassTable(i,3)
            class = i;
            break;
        end
    end
    if class == 0
        continue;
    elseif classesLeft(class) == 0
        continue;
    end

    inds_index = Class(:,3) == class;
    inds = Class(inds_index,1);

    for i=inds'
        if indLeft(i) ~= 0
            indsAdded = indsAdded+1;
            if i <= nrParents
                Pop(:,:,indsAdded) = Parents(:,:,i);
            else
                Pop(:,:,indsAdded) = Children(:,:,i-nrParents);
            end
            indLeft(i) = 0;
            if i == inds(length(inds))
                classesLeft(class) = 0;
            end
            break;
        end
    end
    if indsAdded == nrParents
        break;
    end
end

end
```