Alternative computer architectures and programming languages report: Prolog

# Jigsaw Puzzle Solver

Lu Knoblich

## Table of contents

```
@startuml

:**solutionDownwards**;
    if (previous row __is__ last bottom row) then (no: continue)
        :- matchRight starting on left margin
          matching upper row shape
        - add pieces from new row
          to list of unavailable pieces
        - solutionDownwards below this new row (recursion)
        **return** this row and following rows;
    else (yes: stop)
        : no more rows can be built (puzzle complete)
        **return**  empty list;
    endif
:- list containing current and following rows;
@startuml
```

```
@startuml

:**matchRight**;
    if (current piece == right margin) then (no: continue)
        :- get required Left and Upper shapes
        - search piece with matching shapes
        - check if piece is still available
        - matchRight on new piece (get following matches)(recursion)
        **return** this piece and following matches;
    else (yes: stop)
        :**return**  empty list;
    endif
:- list of pieces forming built row
- bottom shape of built row;
@startuml
```

```
@startyaml
pieceSearch(Piece, [Upper, Right, Down, Left]):
    rotation 0 deg: piece(Piece, [Upper, Right, Down, Left])
    rotation 90 deg: piece(Piece, [Left, Upper, Right, Down])
    rotation 180 deg: piece(Piece, [Down, Left, Upper, Right])
    rotation 270 deg: piece(Piece, [Right, Down Left, Upper])

@endyaml
```

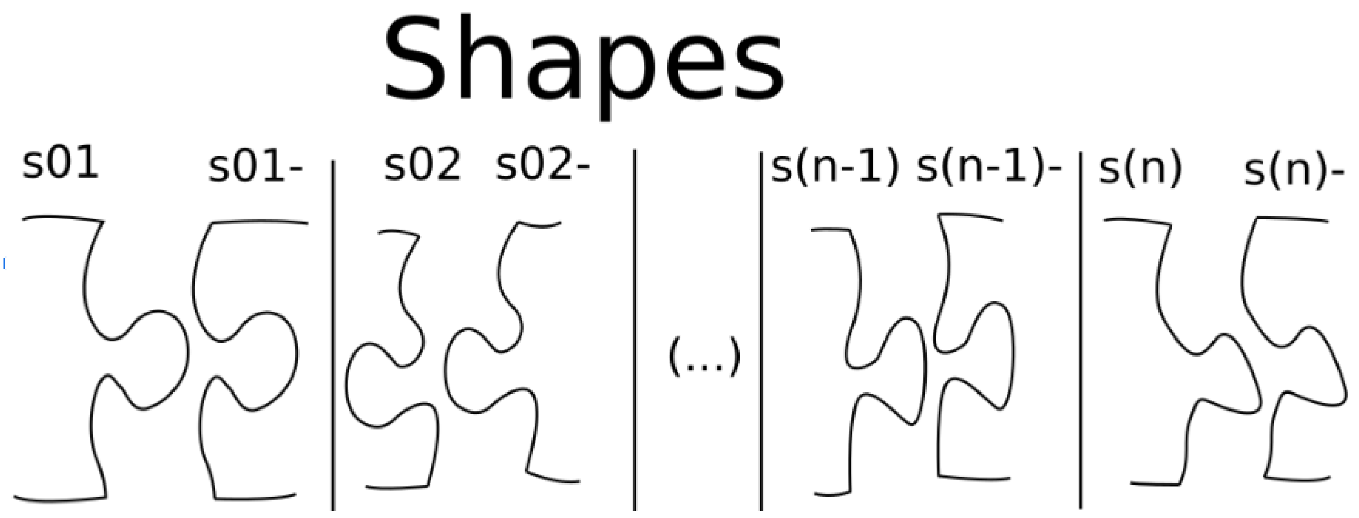
## Description of problem and scope of the solution

For this Project of the "Alternative computer architectures and programming languages" course, the goal was to solve a problem using Prolog.

I have decided to create a Prolog program that is able to solve jigsaw puzzles of any dimensions relying exclusively on the shape of the pieces to find matches that can be connected to built the complete rectangular puzzle.

The creation of the puzzle will be performed using a python script that divides a rectangle into an X by Y matrix, where every two neighboring cells share a matching pair of shapes on the connecting sides. A pair of shapes is said to match if one is the inverse or complementary shape of the other.

In a real puzzle, this can be understood with the help of the following image, where any shape is has the inverted form of its matching shape.



The shape identification of the pieces of a real puzzle, would require an image of the pieces and for the edges to be analyzed as a curved edge to find a matching shape. This was considered to be outside the scope of this project. Instead, the exact shape of each piece will be assumed to be previously known, and each piece's side will be defined to a value from a discrete set of possible shapes, where each of them has a unique corresponding matching shape.

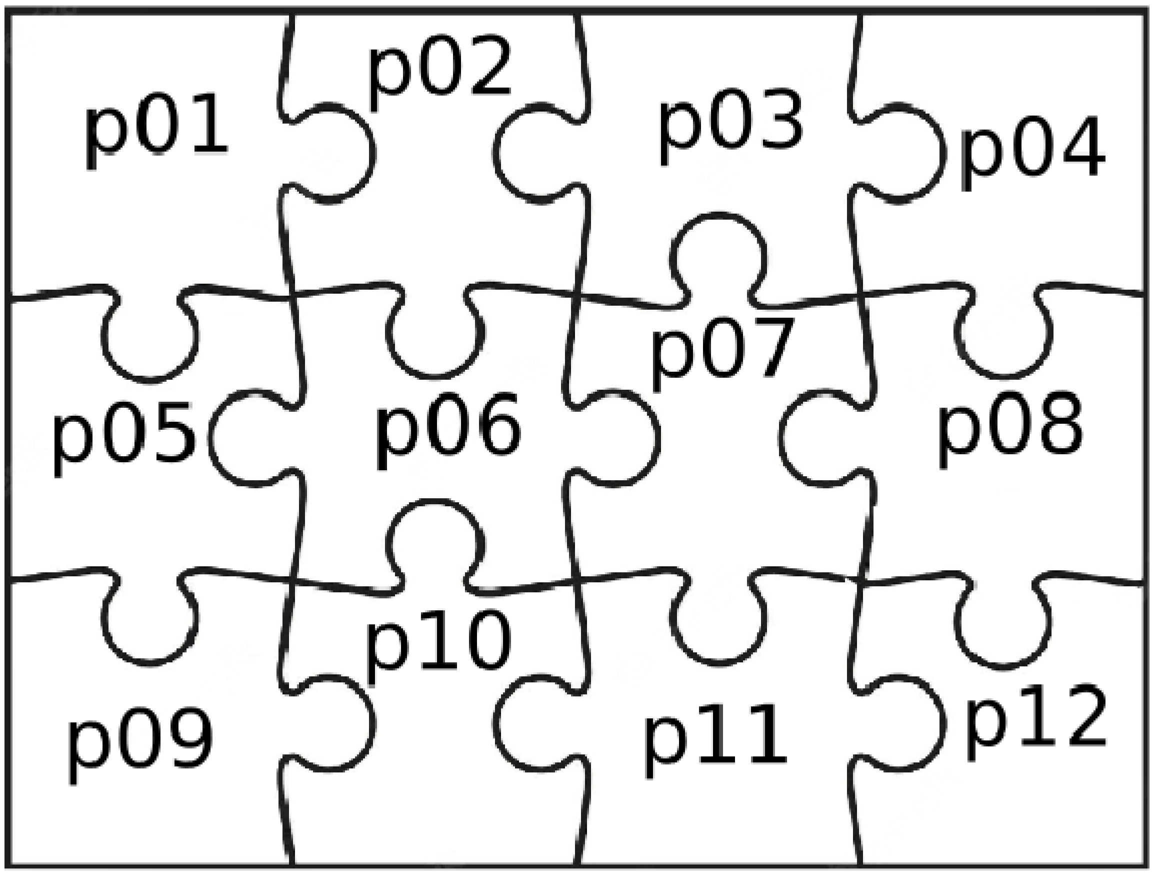
The number of possible shapes each piece's side can have will be limited as a function of the total number of pieces that form the final puzzle. Special consideration needed to be taken in order to define this.

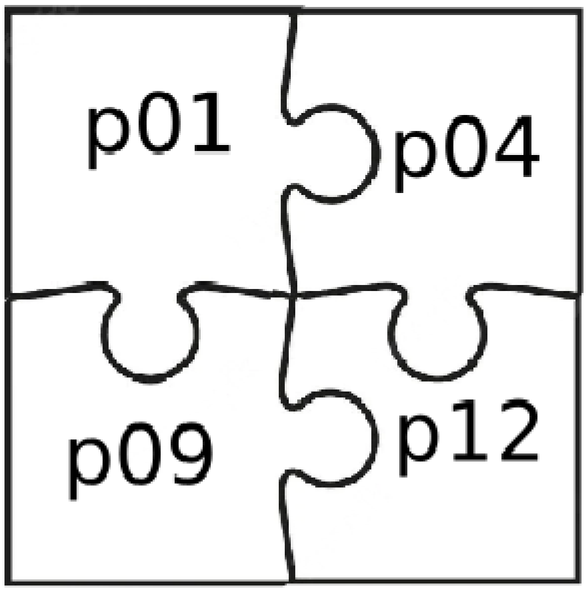
If the number of possible shapes is too large, the amount of pieces that could match to another piece shape will be reduced down to being a single one. This would make the search for a puzzle solution trivial, since little to no backtracking will be necessary, as the search for a piece with the required shapes would potentially yield a single possible solution.

This was deemed unrealistic, since one of the inherent challenges posed by a jigsaw puzzle is having to backtrack a few steps due to finding a matching piece and later discovering that it produces a built section that can not be expanded to completion.

The opposite is also problematic. If the set of possible shapes is too low, there would be an increased number of possible solutions that would yield a rectangular built puzzle with the pieces in different locations, or only utilizing a subset of the available pieces.

An extreme example of this would be a puzzle of 3x3 or larger where all four corner match each other to produce a built rectangle of 2x2 pieces.





A preliminary decision was made to use a 4x5 puzzle with a set of 10 possible matching shapes (20 in total. 10 shapes + 10 inverted shapes) during the development of this solver. This was later evaluated and the set of matching shapes was defined to be one 6th of the total number of pieces in the complete puzzle.

## Solution Procedure

The basic Procedure in order to find a solution is as follows:

The puzzle will be built one piece at a time from left to right to form each row, with the rows being built top to bottom.

To begin each row, the connecting shape of the previous row is needed, and the current piece is set to be the left margin (starting piece). In the case of the first row, the shape of the upper row is set to be that of the margin.

From the current piece, a new piece matching on its right side while also matching the first connecting shape from the row above is searched. Once it is found, this step is repeated recursively with the new piece as the current piece and the remaining shape of the previous row.

The end of this recursion is reached when the current piece is part of the margin, indicating the end of the row, and returning an empty list of pieces.

This is followed by each level of the recursion returning a list with its "current piece" on the front, followed by the list of pieces returned from the deeper recursion levels. The lower shapes of each piece is returned in the same manner.

This results in a list containing the built row, and the exposed shapes needed to connect the next row.

When a row is built, this procedure is repeated recursively to obtain the row below the current one.

The final row of the puzzle is detected when the bottom shape of the previous row matches the margin. This returns an empty list of rows.

Each recursion returns a list containing its "current row" followed by the rows below.

## each recursion to find the next piece is started with left margin as the current piece

## each recursion to build the next row is started with the top margin as the previous/connecting row

## Predicates

The proposed procedure needs to be implemented in prolog using different predicates.

These predicates can be classified in two different groups according to their purpose: - Base predicates - Functional predicates

### Base Predicates

These predicates are the ones used to define the individual puzzle pieces, the matching edges and the puzzle margins and to be able to search for them

**piece/2**

The pieces are declared in the following manner:

```
```

```
piece(p01, [s00,s01,s02n,s00]).
% (...)

piece(marginLeft,[null,smargin,null,null]).
piece(margin2d,[smargin,smargin,smargin,smargin]).
piece(margin2d,[s00,smargin,smargin,smargin]).
```

This example starts with the definition of the upper left corner piece, where "p01" is the piece's name and is followed by the shape of its sides in the following order: Up, Right, Down and Left.

It is followed by the definition of the puzzle's left margin, the upper and lower margin, and lastly, the right margin. These had to be separated in order avoid matching two margins pieces to one another when searching for a new piece, except for the right margin which may match below another margin piece, as it is required to trigger the end of a row.

#### shapeMatch/2 and matchingShapes/2

```
shapeMatch(s00, smargin).
shapeMatch(s01, s01n).
shapeMatch(s02, s02n).
% (...)

matchingShapes(A, B) :- shapeMatch(B, A).
matchingShapes(A, B) :- shapeMatch(A, B).
```

The shapes with their corresponding match are defined using `shapeMatch/2`. To indicate that one shape is the inverse of the other, both will be named equally except for one of them ending with the letter `n` (standing for negative). Nevertheless, this match may not be unidirectional, so that either one of the pieces forming a match can be used to find the second one.

This is achieved through the predicate `matchingShapes/2`, which was kept as a separate predicate to avoid the endless loops that would occur if `shapeMatch(A,B):- shapeMatch(B,A).` had been used.

#### pSearch/2 and pieceSearch/3

While the individual pieces can be found using the piece/2 predicate, this would lead to them having a fixed orientation. If it were a real puzzle, it would be like all pieces having an arrow that is known to point upwards. This was considered to be an unrealistic representation of the problems posed by a jigsaw puzzle.

To allow the pieces to be randomly oriented and require the solver to "rotate" the pieces to find match, the following predicates were implemented:

```
pSearch(P,[U,R,D,L]):-
    between(0,3,N),
    pieceSearch(P,[U,R,D,L],N),
    P \= margin2d,
    P \= marginLeft.

% margin Pieces should not be rotated
pSearch(margin2d,[smargin,smargin,smargin,smargin]).
pSearch(margin2d,[s00,smargin,smargin,smargin]).
pSearch(marginLeft,[null,smargin,null,null]).

% different rotations
pieceSearch(P,[U,R,D,L],0):- piece(P,[U,R,D,L]).
pieceSearch(P,[U,R,D,L],1):- piece(P,[L,U,R,D]).
pieceSearch(P,[U,R,D,L],2):- piece(P,[D,L,U,R]).
pieceSearch(P,[U,R,D,L],3):- piece(P,[R,D,L,U]).
```

`pSearch` is responsible for attempting all possible orientations for pieces of the puzzle without rotating the margin pieces, since they need to be identified to allow the beginning and end conditions of each row and the complete puzzle.

`pieceSearch` declares the 4 different orientations each piece can have by rolling the shapes of the edges.

### Functional Predicates

`colset STAFF = with emp0 | emp1 | emp2 timed;` This color set represents the workers, or employees. The names are used interchangeably. There is no specific need to vary between the employees. One could either have one of each or three of the same employee. It will not make a difference for the process inside the CPN, however it can help follow the exact paths the employees take.

`colset CHEMTYPE = with dev | fix | bleach | flo | noChem;` This color set describes the various types of chemicals used in the different steps of the development process. The `noChem` type is included to account for the absence of a chemical. As will be seen later, this can also be used to describe a chemical which has lost its potency.

```
colset EXTRATIME = INT;
colset TAKEOUTTIME = INT;
colset PROCESSTIME = INT;
```

These three color sets describe specific timings. `EXTRATIME` is the amount of time spent inside chemical tanks which exceeds the targeted process time. `TAKEOUTTIME` is the timestamp (in simulator steps) at which a roll inside a tank is finished processing and should be taken out of it to obtain optimal results. The `PROCESSTIME` is the amount of time (in simulator steps) that a process takes.

`colset ROLL = product ROLLTYPE * EXTRATIME;` Rolls are the product of a specific `ROLLTYPE` and the accumulated amount of `EXTRATIME` spent inside the chemical tanks. This creates an intrinsic state which would be analogous to the quality of the picture in the real world domain.

`colset CHEMTYPExPTIME = product CHEMTYPE * PROCESSTIME;` This color set is a product of chemical types and the respective amount of time each roll needs to spend in the tank containing this chemical.

`colset CHEMBOTTLE = product CHEMTYPE * INT;` The `CHEMBOTTLE` color set is used in the chemical storage and describes how many film rolls can be processed by one instance of this chemical bottle. Even though it is not used in this example, it can even be used to model bottles with partially exhausted chemicals.

`colset ROLLxCHEM = product ROLL * CHEMBOTTLE * TAKEOUTTIME timed;` This is the most extensive color set, combining almost all of our previous color sets. This is used to model the state markings of our chemical tanks.

## Points of Interest in the model

The main points of interest inside the Model are the Chemical tanks. The sub-page for a chemical tank looks like this: Screenshot of a chemical tank

### Processing film rolls

The new rolls enter from an arbitrary input and need to be put into the chemical tank. For this to happen a few conditions have to be met. An Idle employee needs to be present, a specified chemical type along with its specific process time needs to be fed into the process, the chemical tank must not contain another roll already and the chemical needs to be potent enough to process another film roll. Once the conditions have been met the rolls can be put into the chemical tank. This is performed while calling the `startProc(roll: ROLL,(chemType,n): CHEMBOTTLE, pt: PROCESSTIME)` function and sending the employee back to idling with a variably increased timestamp. Once the chemical process is completed, the rolls can be taken out of the chemical tank. Again, there is a need for an employee to facilitate the process, which takes a varying amount of time. Ending this transition is the `stopProc(roll: ROLL, tot:TAKEOUTTIME)` function.

```
fun startProc(roll:ROLL,chembottle:CHEMBOTTLE,proc_time:INT): ROLLxCHEM =
(*Sets the time at which the roll should be taken out*)
let
    val takeouttime = intTime()+proc_time
in
    ((roll,chembottle,takeouttime))
end
```

The `startProc` function calculates the time at which the film roll should be taken out of the chemical tank and writes it into the colorset.

```
fun stopProc((rolltype,extratime):ROLL,tot:TAKEOUTTIME): ROLL =
(*If roll stayed in tank for too long, the extra time gets added to
extratime*)
let
    val timeTooLate = intTime()-tot;
    val new_extratime = extratime+timeTooLate;
```

```
in
    ((rolltype,new_extratime))
end
```

The `stopProc` function calculates the difference between current time and the time the roll should have been taken out and adds it to the extratime variable stored inside the `ROLL` color set.

### Refilling the chemical tanks

Refilling the chemical tanks works by taking a `CHEMBOTTLE` out of the Chemical storage and putting the new, fresh solution into the tank. The transition is facilitated by an idle employee and the knowledge of which chemical is supposed to be put in the tank. The exhausted chemicals that were in the tank are discarded into the chemical waste. This process also takes up a varying amount of time for the worker.

The time variance has been implemented using three parts.

Firstly a static variance has been defined: `val variance = n (*in the actual simulations an Integer was provided*)`.

Secondly a function which delivers a discrete variation of values has been implemented:

```
fun DELAY(max:INT) = discrete(0,max);
```

Lastly these two have been combined in the vary function like this:

```
fun vary(n: INT) = (n-variance)+DELAY(2*variance);
```

This allows for calling the `vary` function on any integer which returns a value somewhere between the initial value minus the variance and the initial value plus the variance.

A concrete example would be setting `variance = 2` and calling `vary(20)`, which returns a value between 18 and 22.

In other words, tailored for our specific model, an action taken by an employee that would normally take 2 minutes can now take anywhere between 1:48 minutes or 2:12 minutes.

This allows the inspection of the resilience of the model in regard to variance in timings, through multiple iterations, since the variance does not change during a given simulation.

### Sorting

In the end the rolls are sorted using the `isTrash` function. An arbitrary measure of quality has been established which states that if a roll has been processed in the chemicals for one minute longer than it should have been, it won't produce acceptable pictures during the scanning or enlargement process. This includes the accumulated amount of extra time through all tanks. A negative implementation was omitted since one could theoretically always put the roll in question back into the chemical tank.

The concrete implementation of said function is:

```
fun isTrash(extratime:EXTRATIME)=
let
  val isTrash: BOOL = extratime > 9;
in
  isTrash
end
```

In the context of this model means if a film roll has been processed for an additional minute or longer it will be deemed unusable.

## Results of the model simulation and analysis

The results of the simulation include a State space analysis, which was mostly unsuccessful due to the high computation times, and a quantitative analysis of the average extra time on rolls which exit the process. This analysis has been performed on both the rolls which are thrown away and the ones which pass the quality test and are sent to scanning. However, the data sets have been recorded and evaluated separately.

### State space analysis

#### One employee, one roll

state space nodes and arcs. one employee, one film roll

| Variance | Nodes | Arcs |
|---|---|---|
| 0 | 171 | 225 |
| 1 | 1007 | 1103 |

| Variance | Nodes | Arcs |
|---|---|---|
| 2 | 1476 | 1594 |

As can be seen the introduction of variance vastly increases the number of nodes and arcs of the state-space graph.

#### One employee, two rolls

state space nodes and arcs. one employee, two film rolls

| Variance | Nodes | Arcs |
|---|---|---|
| 0 | 881 | 1019 |
| 1 | 11051 | 11795 |
| 2 | 16628 | 17677 |

Adding a second roll increases the amount of nodes and arcs in the state-space graph tenfold. The relative increase due to variance stays fairly constant.

#### One employee, three rolls

state space nodes and arcs. one employee, three film rolls

| Variance | Nodes | Arcs |
|---|---|---|
| 0 | 2840 | 3161 |
| 1 | 64604 | 69201 |
| 2 | 104106 | 111000 |

Again, adding a third roll increases the amount of nodes and arcs tenfold to the previous example. The relative increase due to variance still stays fairly constant.

### Monitors, success rates and evaluation of extra time(s)

Two data collection monitors have been implemented, which both collected the extra time on the rolls which left the laboratory.

The monitor logic worked the same in both cases, therefore only one will be presented here.

The predicate:

```
fun pred (bindelem) =
let
  fun predBindElem (Main_Process'throw_into_trash (1,
                                 {et,rt})) = true
      | predBindElem _ = false
in
  predBindElem bindelem
end
```

Due to the `isTrash` function we did not need to sort for specific elements, therefore the predicate always returns true for the given transition.

The observer:

```
fun obs (bindelem) =
let
  fun obsBindElem (Main_Process'throw_into_trash (1, {et,rt})) = et
      | obsBindElem _ = 0
in
  obsBindElem bindelem
end
```

The observer extracts the extra time out of our `ROLL` color set and stores its value and averages it out over the duration of the simulation.

The results were three data sets.

The first one describes the success percentage (amount of rolls that go into scanning divided by the amount of rolls that entered the process) based on variance and the amount of employees present inside the laboratory:

| **Number of employees** | **success rate with variance=0** | **success rate with variance=1** | **success rate with variance=2** |
|---|---|---|---|
| 1 | 16.38% | 13.24% | 12.88% |
| 2 | 74.64% | 75.34% | 75.48% |
| 3 | 98.10% | 99.10% | 98.90% |
| 4 | 98.96% | 99.89% | 99.70% |
| 5 | 99.98% | 99.98% | 99.98% |
| 6 | 100.00% | 100.00% | 100.00% |

Success percentage based on number of employees and variance

It is immediately noticeable that a single employee has very poor chances of handling multiple film rolls simultaneously. With a growing number of employees the curve starts to approach 100% logarithmically.

Interestingly the variance has a (slightly) negative effect for a low number of employees but a (slightly) positive effect for higher numbers of employees.

At 6 employees the success rate has reached 100%.

---

The second data set describes the average amount of extra time on rolls which pass the quality test:

| **Number of employees** | **avg extra time at variance=0** | **avg extra time at variance=1** | **avg extra time at variance=2** |
|---|---|---|---|
| 1 | 3.71 | 4.75 | 4.67 |
| 2 | 2.2 | 3.06 | 3.13 |
| 3 | 0.72 | 0.84 | 0.85 |
| 4 | 0.12 | 0.34 | 0.37 |
| 5 | 0.07 | 0.07 | 0.07 |
| 6 | 0 | 0.01 | 0 |

Average extra time on rolls ready for scanning

It is noticeable that the variance greatly affects the average extra time negatively. It is especially noteworthy, that the graphs with the variance of 1 and 2 diverge further from the graph with variance 0 between 3 and 4 employees, before converging again and even improving over variance 0 at 5 employees.

---

The third and last data set describes the average amount of extra time on rolls which **do not** pass the quality test:

| **Number of employees** | **avg extra time at variance=0** | **avg extra time at variance=1** | **avg extra time at variance=2** |
|---|---|---|---|
| 1 | 37.67 | 38.68 | 38.57 |
| 2 | 14.93 | 15.73 | 16.13 |
| 3 | 13.47 | 14.49 | 13.9 |
| 4 | 10 | 10.67 | 10.31 |
| 5 | 10 | 11 | 10 |
| 6 | 0 | 0 | 0% |

|Average extra time on rolls to be thrown away

Here all graphs follow a similar line with a slight offset until they converge at 0 for 6 employees, since they are always able to properly develop any amount of film rolls which enter the lab. It should be noted that in the beginning the higher variance lead to worse values for average extra time, as one might expect, until it improves at the point of 3 employees and even improves over its counterpart with a variance of 1.

## Look back: Did CPN fit to the miniproject and its subject? What was good, what was bad?

CPN is a great tool to visualize processes and determine strong connections between colorsets inside the Net. However, inspecting, analyzing and evaluating those processes can be quite tedious as running a simulation with mediocre complexity can already lead to multiple hours of calculating state-spaces. Since the generated datasets aren't very large (contrary to what might be expected based on the computation time) it can be even more frustrating. The usability of CPN Tools also posed some challenges. While the approach of binding tools to the cursor and having a contextual radial menu was refreshing and led to a unique experience developing the CPN, it was somewhat overshadowed by the lack of (now) common possibilities when it comes to editing text (like holding Ctrl to skip whole word, holding Shift to highlight things the cursor moves over and many other). Due to the fact that CPNs and CPN Tools were far ahead of their time when they were developed this shall not be held against them. An update would most definitely lead to a wider increase in use.