Animation of the N-Queens Problem in JavaScript

Study Report

from the Course of Studies "Applied Computer Science"

at the Cooperative State University Baden-Württemberg Mannheim

by

Koen Loogman & Jessica Roth

29. April 2019

Time of Project	17. September 2018 to 29. April 2019
Student ID, Course	4867747 & 5188586, TINF16AI-BC
Tutor	Prof. Dr. Karl Stroetmann

Formalities

Declaration of academic sincerity

Hereby I solemnly declare:

1. that this Study Report, titled *Animation of the N-Queens Problem in JavaScript* is entirely the product of my own scholarly work, unless otherwise indicated in the text or references, or acknowledged below;
2. I have indicated the thoughts adopted directly or indirectly from other sources at the appropriate places within the document;
3. this Study Report has not been submitted either in whole or part, for a degree at this or any other university or institution;
4. I have not published this Study Report in the past;
5. the printed version is equivalent to the submitted electronic one.

I am aware that a dishonest declaration will entail legal consequences.

Mannheim, 29. April 2019 — Location, date

Signature Koen Loogman & Jessica Roth

Abstract

Animation of the N-Queens Problem in JavaScript

This work is about the animation of the Davis-Putnam algorithm solving the N-Queens Problem in JavaScript. Starting with the scientific basics needed to understand the algorithm and said problem, we continue with some technical basics regarding JavaScript that will affect the implementation. With the foundation set, we continue with a conception and the realization in JavaScript. At the end we evaluate the result, making a conclusion and looking ahead to possible further improvements or applications of said result.

Animation des N-Damen Problems in JavaScript

Diese Arbeit befasst sich mit der Animation des Davis-Putnam Algorithmus beim Lösen des N-Damen Problems in JavaScript. Zuerst werden die wissenschaftlichen Grundlagen, die für das Verständnis des Algorithmuses und dem genannten Problem benötigt werden, vermittelt. Dazu kommen noch ein paar technische Grundlagen, die für die Implementierung in JavaScript relevant sind. Nachdem alle Grundlagen bekannt sind, wird zuerst ein Konzept erstellt, welches dann bei der Implementation befolgt wird. Das Ergebnis daraus wird zum Schluss evaluiert und bezüglich möglicher Verbesserungen und anderen Verwendungen betrachtet.

Contents

List of Figures

1 Introduction

1.1 Motivation

The *Davis-Putnam algorithm* is an important tool for solving known problems from propositional logic such as the *N-Queens Problem*. This is also the reason why it is often used to explain propositional logic principles. However, it is often difficult to understand how the algorithm finds possible solutions for propositional logic problems, as it is difficult to understand by just looking at the algorithm's computational paths.

If there would be the possibility to follow the *Davis-Putnam algorithm* trying to find a solution for the *N-Queens Problem* step by step with a visual representation of the current state, then this would clarify many understanding problems and enhance the learning process. This is also the reason why this program is so important for visualizing the Davis-Putnam algorithm. It can be used for many purposes, such as supporting lectures, and thus helps many interested people to easily get started with this complex algorithm.

1.2 Objective of the work

The task of this student research project is the animation of the *Davis-Putnam algorithm* on the basis of the *N-Queens Problem* with the help of JavaScript. Several criteria are defined, which this animation has to fulfill, since it serves above all for the visualization and for the more exact descriptive clarification and/or explanation of the algorithm. The exact requirements are specified more precisely in the course of this work and evaluated at the end of the final product. The main part of this work therefore deals with the conception and implementation of this task.

1.3 Structure of the report

In order to facilitate navigation through this work, the following section takes a brief look at the structure of this work.

The next chapter explains the scientific basics of the *Davis-Putnam algorithm* and the *N-Queens Problem*. These are mainly about the logical correlations and the mathematically correct definition of the problem to be solved.

The third chapter then gives a general overview of the libraries and technologies used and provides background information on the reasons why certain technologies were preferred or selected.

The fourth section of this paper takes a closer look at the design of the program. It mainly deals with the general architecture and the created layout design, which serves as a template for the implementation of the user interface. The chapter is introduced by specifying the requirements for the program, which are mandatory for the evaluation at the end.

In the fifth chapter, step by step you will be guided through the finished implementation and the structure of the program will be explained in more detail. This chapter guides through the different components and classes and explains their roles in more detail.

The last chapter starts with the evaluation of the requirements. This is mainly to see if the program meets all the criteria and can be considered successful at the end. This is followed by a summary and an outlook on how the program or its components could be further developed or used in other ways in the future.

2 Scientific Basics

The purpose of this work is to visualize the *Davis-Putnam algorithm* solving the so-called *N-Queens Problem*. Therefore a general understanding of this algorithm and the mathematical problem has to be created.

This chapter summarizes the fundamental knowledge that is needed in order to create a basis for further development. Among other things, the definition of the mathematical problem plays a role here, so that a possible solution can be found by the *Davis-Putnam algorithm*.

2.1 Basics of Propositional Logic

Before the algorithm can be examined in more detail, a few mathematical terms must first be briefly explained.

Literal A propositional logic formula f is called a literal, if one of the following cases is true.

1. $f = \top$ or $f = \bot$
2. $f = p$, where p is a propositional variable. If this is the case, it is called a positive literal.
3. $f = \neg p$, where p is a propositional variable. Then it is called a negative literal.

The set of all literals is called $\mathcal{L}$.

Clause A propositional logic formula C is called a clause if it has the following structure.

$$C = l_1 \vee \cdots \vee l_r$$

Here, l_i is a literal for every value $i \in \{1, \ ..., \ r\}$.

The set of all clauses is called $\mathcal{K}$.

Set of propositional logical formulas Propositional logical formulas are words, which are created from the alphabet $\mathcal{A} := \mathcal{P} \cup \{\top, \bot, \neg, \vee, \wedge, \rightarrow, \leftrightarrow, (,)\}$
The set of propositional logical formulas $\mathcal{F}$ is defined by induction:

1. $\top \in \mathcal{F}$ and $\bot \in \mathcal{F}$
   Here, $\top$ is called Verum and stands for the formula, which is always true. $\bot$ is called Falsum and stands for the formula, which is always false.

2. If $p \in \mathcal{P}$, then $p \in \mathcal{F}$ also applies.
   Every propositional logical variable is also a propositional logical formula.

3. If $f \in \mathcal{F}$, then $\neg f \in \mathcal{F}$ also applies.
   The formula $\neg f$ is called the negation of f.

4. If $f_1, f_2 \in \mathcal{F}$, then
   $(f_1 \vee f_2) \in \mathcal{F}$ (Disjunction of f_1 and f_2)
   $(f_1 \wedge f_2) \in \mathcal{F}$ (Conjunction of f_1 and f_2)
   $(f_1 \rightarrow f_2) \in \mathcal{F}$ (Implication of f_1 and f_2)
   $(f_1 \leftrightarrow f_2) \in \mathcal{F}$ (Biconditional of f_1 and f_2) also applies.

The set $\mathcal{F}$ is the smallest subset of words created from alphabet $\mathcal{A}$, which has the previous listed characteristics.

Conjunctive normal form A formula F is in conjunctive normal form if and only if F is a conjunction of clauses:

$$F = K_1 \wedge \cdots \wedge K_n,$$

where K_i is a clause for all $i = 1, \cdots, n$.

Set of propositional logical variables A set $\mathcal{P}$ is defined as the Set of propositional logical variables. Typically the set consists of small types from the Latin alphabet, which also may be indexed.
For example: $\mathcal{P} := \{\ p,\ q,\ r,\ p_1,\ p_2,\ p_3\ \}$

Propositional logic interpretation The function $\mathcal{I} : \mathcal{P} \rightarrow \mathbb{B}$ is called propositional logic interpretation, which assigns a true value $\mathcal{I}(p) \in \mathbb{B}$ to each proposition-logical variable $p \in \mathcal{P}$

Satisfiable There is set of propositional logic formulas K. If there is an interpretation K so that all formulas of the set K are satisfied and the formula $\mathcal{I}(f) = \texttt{True}$ for all propositional logic formulas $f \in K$ is given, so set K is satisfiable. If this is not the case, then set K is not satisfiable.

Tautology A propositional logic formula f is a tautology if and only if $\mathcal{I}(f) = \texttt{True}$ for every interpretation $\mathcal{I}$. In this case it is written as $\models f$.

Solution If there is an interpretation $\mathcal{I}$ that satisfies a set of propositional logic formulas K, then $\mathcal{I}$ is a possible solution of K.

2.2 The Algorithm of Davis-Putnam

The *Davis-Putnam algorithm* is a method for calculating a solution of a set of propositional clauses. With clause sets that contain just one literal per clause this can be easily determined, as can be seen in the following two examples.

$$K_1 = \Big\{ \{r\},\ \{\neg s\},\ \{t\},\ \{\neg u\},\ \{\neg v\} \Big\}$$

K_1 can also be written as a propositional logic formula.

$$r \wedge \neg s \wedge t \wedge \neg u \wedge \neg v.$$

It is obvious that this formula is solvable by r and t "true" and s, u and v having the value "false". As a negative example we consider the set K_2, which is defined as follows:

$$K_2 = \Big\{ \{r\},\ \{\},\ \{t\} \Big\}$$

K_2 can also be written as a propositional logic formula.

$$r \wedge \bot \wedge t.$$

The empty set in the set of clauses K_2 is interpreted as a $\bot$ in the propositional logic, making it impossible to satisfy K_2. The set of clauses K_1 can easily be satisfied, since it only contains clauses with a single literal. Once the clauses of K_1 contain more literals it gets more complex to satisfy. For these algorithms like the *Davis-Putnam algorithm* are used. But before we get to the details, two definitions have to be introduced first [1].

Unit clause A clause C is a unit clause if it contains of only a single literal, i.e. a variable or a negated variable.

Trivial clause sets A set of clauses is trivial if and only if one of the following two cases occurs.

1. A set of clauses K contains the empty set and is therefore unsatisfiable.
2. The unit clauses always contain different propositional variables, so that only the clause $\{p\}$ or $\{\neg p\}$ can occur. If this is the case, a solution for the clause set can be determined.

In order for one of these two cases to occur, the clause sets must be simplified with the help of the following three options so that they consist only of unit clauses.

1. Cut Rule
2. Subsumption
3. Case Differentiation

2.2.1 Simplification with the Cut Rule

To simplify a set of clauses K with $C_1 \cup \{ p \}, C_2 \cup \{ \neg p \} \in K$ the cut rule can be used. Let's say we have two clauses C_1, C_2 and a variable p, then a typical application of the cut rule is.

$$\frac{C_1 \cup \{ p \} \qquad C_2 \cup \{ \neg p \}}{C_1 \cup C_2}$$

In this case the result clause will in general have more literals than $C_1 \cup \{ p \}$ or $C_2 \cup \{ \neg p \}$ alone. Because if the clause $C_1 \cup \{ p \}$ contains $m+1$ literals and $C_1 \cup \{ p \}$ contains $n+1$ literals, then the union $C_1 \cup C_2$ can have upto $m+n$ literals in total. In general $m+n$ may be bigger than $m+1$ or $n+1$. The clause can be smaller than previously if both sets contain the same literals, but it is only granted to be smaller if $m = 0 \vee n = 0$. This is the case for unit clauses. Therefore we will only allow the use of the cut rule if one of the clauses is a unit clause. These cuts will be called unit cuts. To be able to do those unit cuts with a unit clause $\{ l \}$ on all clauses of a set of clauses K, we define the function

$$\texttt{unitCut} : 2^K \times L \to 2^K$$

so that for a set of clauses K and a literal l the function $\texttt{unitCut}(K, l)$ will simplify the set of clauses as much as possible by using unit cuts:

$$\texttt{unitCut}(K, l) = \{ C \setminus \{ \bar{l} \} \mid C \in K \}$$

The result set of clauses will have the same amount of clauses as K. Only the clauses $C \in K$ where $\bar{l} \in C$ were affected and got that element removed [2].

2.2.2 Simplification with Subsumption

For further simplification of the set of clauses we will use subsumption. The idea is simple and can be shown with the following example:

$$K = \{ \{ p, \neg q \}, \{ p \} \} \cup M$$

It is clear that the clause $\{ p \}$ implies the clause $\{ p, \neg q \}$, because if $\{ p \}$ is satisfied, then $\{ p, \neg q \}$ will be satisfied too. That's because

$$\models p \rightarrow p \vee \neg q$$

is given. With that in mind we can say that we can subsume a clause C with a unit clause U if $U \subseteq C$. That means if we have a set of clauses K with $C \in K \wedge U \in K$ and U subsumes C, we can remove the clause C from K to reduce its size. We define a function

$$\texttt{subsume} : 2^K \times L \rightarrow 2^K$$

that receives a set of clauses K and a literal l. It will simplify K by subsuming with the unit clause $\{ l \}$:

$$\texttt{subsume}(K, l) = \{ C \in K \mid l \notin C \} \cup \{ \{ l \} \}$$

We have to add the unit clause to the set of clauses, because $l \in U$ is the case. Resulting in U being removed from K at first too [2].

2.2.3 Simplification with Case Differentiation

The following theorem forms the basis for the principle of case differentiation.

Theorem The set of clauses K can only be satisfied if and only if $K \cup \Big\{ \{p\} \Big\}$ or $K \cup \Big\{ \{\neg p\} \Big\}$ can be satisfied.

For simplification, a propositional variable p is selected at the beginning, which occurs in the clause set. Then the two clause sets mentioned above are formed and an attempt is made to find a solution for one of them. If this attempt is successful, the result is automatically a solution of K. If none is found, K is not satisfiable.

2.2.4 The Davis-Putnam Method

The theoretical foundation that was previously created now makes it possible to sketch the procedure of the *Davis-Putnam algorithm*. With the help of the intersection rule and the subsumption, the clause set K is simplified as far as possible. If already after this

step K is trivial, the procedure is finished. Otherwise a propositional logical variable p is selected, which occurs in K. Then a recursive attempt is made to solve the clause set $K \cup \left\{\{p\}\right\}$ in order to find a solution for K. If no solution was found here either, the same is tried with the negated p. If this attempt also fails, K is unsatisfiable. A possible recursive implementation of the method can be seen in listing 1 [1], [2].

```
function Satisfiable( clause set S ) return boolean
    /* unit propagation */
    repeat
        for each unit clause L in S do
            /* unit subsumption */
            delete from S every clause containing L
            /* unit resolution */
            delete L̄ from every clause of S in which it occurs
        od
        if S is empty then
            return true
        else if a clause becomes {} in S then
            return false
        fi
    until no further changes result
    /* splitting */
    choose a literal L occurring in S
    if Satisfiable (S ∪ {{L̄}}) then
        return true
    else if Satisfiable (S ∪ {{L}}) then
        return true
    else
        return false
    fi
end function
```

Listing 1: A simple Davis–Putnam algorithm [1]

2.3 N-Queens Problem

The N-Queen problem is about how to place n queens on an $n \times n$ chessboard so that none would be obstructed in their turn. A special example would be the 8-Queens Problem, which is related to the standardized chessboard. A queen in a normal game of chess can move diagonally, vertically and horizontally. This move pattern can be seen in Figure 2.1. In summary, this means that there is only one queen allowed on her vertical, horizontal and diagonal line at a time so that they do not interfere with each other. In this problem it is assumed that any queen can attack any other queen and the field colors are ignored.

This problem can be solved by several algorithms such as the *Davis-Putnam algorithm* [3], [4] [2].

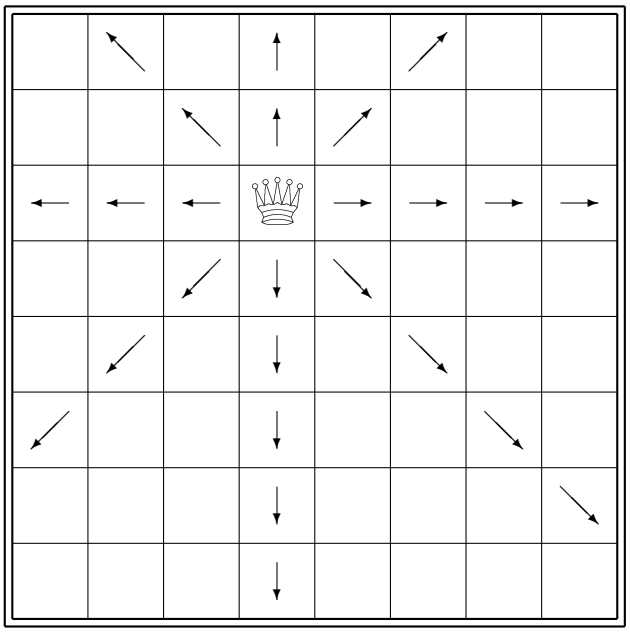

Figure 2.1: Movement pattern of a queen on an 8 by 8 board [2]

2.3.1 N-Queens Problem as Propositional Formula

Before we can find a solution for the *N-Queens Problem*, we need to define the problem as a propositional logic formula. For that we will say that the variable $Q_{x,y}$ is the queen in row x on column y. If for example $Q_{1,2}$ is true a queen will be placed on row 1 and column 2 else not. Now all that is left is the creation of the set of clauses K that describes the *N-Queens Problem*. Therefore we create some functions to help us define the problem as such:

atMostOne The function atMostOne will receive a set of literals L and return a set of clauses K that express the fact that only one of the literals $l \in L$ can be true. We define it to support following functions.

$$\texttt{atMostOne}: \ \mathcal{L} \rightarrow 2^{\mathcal{K}}$$

$$\texttt{atMostOne}(L) := \{ \ \{ \ \neg l_1, \ \neg l_2 \} \ : \ l_1, l_2 \in L \mid l_1 \neq l_2 \ \}$$

This works, because if one variable is said to be true then the other variable must be false, since either one or the other needs to be false.

atMostOneInRow This function receives a number n for the total amount of queens and a number $r \in \{1..n\}$ for the row and return a set of clauses K that express the fact

that only one queen can be placed in row r.

$$\texttt{atMostOneInRow} : \ \mathcal{N} \times \mathcal{N} \to 2^{\mathcal{K}}$$

$$\texttt{atMostOneInRow}(r,\ n) := \texttt{atMostOne}(\{\ Q_{r,i} : \ i \in \{1\ ..\ n\}\ \})$$

oneInColumn This function receives a number n for the total amount of queens and a number c where $c \leq n$ for the column and returns a set of clauses K that express the fact that at least one queen has to be placed in the column c. Because if only one queen can be placed per row, every column needs at least one queen to have n queens on the chessboard.

$$\texttt{oneInColumn} : \ \mathcal{N} \times \mathcal{N} \to 2^{\mathcal{K}}$$

$$\texttt{oneInColumn}(c,\ n) := \{\ \{\ Q_{i,c} : \ i \in \{1\ ..\ n\}\ \}\ \}$$

atMostOneInUpperDiagonal This function returns a set of clauses K that express the fact that only one queen can be placed in the upper diagonal, for a given number n and $k \in \{3\ ..\ 2 * n - 1\}$. The number k helps with calculating which combination of row and column is part of the diagonal.

$$\texttt{atMostOneInUpperDiagonal} : \ \mathcal{N} \times \mathcal{N} \to 2^{\mathcal{K}}$$

$$\texttt{atMostOneInUpperDiagonal}(k,\ n) :=$$
$$\texttt{atMostOne}(\{\ Q_{x,y} : \ x, y \in \{1\ ..\ n\} \mid x + y = k\ \})$$

This works because for an upper diagonal when increasing x by one we reduce y by one and vice versa. So if we create the sum of x and y the result will be the same for each tile of the same diagonal as following equations show.

$$x + y = x + 1 + y - 1$$
$$x + y = x - 1 + y + 1$$

Starting with $k = 3$, because $Q_{1,2}$ and $Q_{2,1}$ are the first upper diagonal, and ending with $k = 2 * n - 1$, because $Q_{n-1,n}$ and $Q_{n,n-1}$ form the last diagonal.

atMostOneInLowerDiagonal This function returns a set of clauses K that express the fact that only one queen can be placed in the lower diagonal, for a given number n and $k \in \{-(n-2)\ ..\ n-2\}$. As before the number k helps with calculating which combination of row and column is part of the diagonal.

$$\texttt{atMostOneInLowerDiagonal} : \ \mathcal{N} \times \mathcal{N} \to 2^{\mathcal{K}}$$

$$\texttt{atMostOneInLowerDiagonal}(k,\ n) :=$$
$$\texttt{atMostOne}(\{\ Q_{x,y} : \ x, y \in \{1\ ..\ n\} \mid x - y = k\ \})$$

Here it is similar to the upper diagonals, but with the change that both x and y either

increase or decrease by one. Now the difference between x and y is the same for each tile of the same diagonal as following equations show.

$$x - y = x + 1 - y - 1 = (x + 1) - (y + 1)$$
$$x - y = x - 1 - y + 1 = (x - 1) - (y - 1)$$

Starting with $k = -(n-2)$, because $Q_{1,n-1}$ and $Q_{2,n}$ are the first lower diagonal, and ending with $k = n - 2$, because $Q_{n-1,1}$ and $Q_{n,2}$ form the last diagonal.

queensClauses With the functions above, we finally can define the complete propositional formula for the *N-Queens Problem* for a given number n in form of a set of clauses K. A set of clauses that express that only one queen can be placed per column is not needed in this case, because that one has to be placed in combination with only one per row implies that only one per column is placed.

$$\texttt{queensClauses} : \mathcal{N} \rightarrow 2^{\mathcal{K}}$$

$$\texttt{queensClauses}(n) := \bigcup_{i=1}^{n} (\texttt{atMostOneInRow}(i,n) \cup \texttt{oneInColumn}(i,n)) \cup \bigcup_{i=3}^{2*n-1} \texttt{atMostOneInUpperDiagonal}(i,n) \cup \bigcup_{i=-(n-2)}^{n-2} \texttt{atMostOneInLowerDiagonal}(i,n)$$

3 Technical Basics

In the previous chapter, the basics of the Davis-Putnam algorithm and the *N-Queens Problem* were analyzed and defined in more detail. Now follows a description of those libraries and technologies that have been used for the implementation. The use of these and the comparison to possible alternatives will be discussed.

3.1 Multi Threading in JS

Many programs and applications, which require many processes to process their functions, are usually faced with a problem. This project, for example, has a high Workload in the background, which is produced by the logical processes of the Davis-Putnam algorithm, but also has to make the changes visible to the user in real time. According to this, the project has to deal with the difficulty that JavaScript is basically a single-thread environment and therefore cannot parallelise scripts. The result would be a slow user interface and constant waiting for the current results. This would significantly worsen the user experience and in the worst case would result in an unusable website.

A solution to this problem is provided by so-called web workers, which are created by an HTML page as background processes apart from the main thread. These can then be used as separate threads, e.g. for computation-intensive scripts, and thus leave the processing of user interactions to the main thread. The workers communicate with the main thread using messages and thus achieve parallelism without creating blocking instances [5], [6].

3.2 Parcel

In order to better understand why the use of a module bundling tool makes sense for the project, a general understanding of the Module Bundling topic must first be created.

Generally speaking, the Node.js Feature module bundling is the merging of a group of modules with their dependencies into a single file. The advantage, especially in web development, is that not every single file in the central HTML file has to be inserted by script tag, but only a single large file. The loading time of the website is significantly improved and it increases maintainability during and after the development process. Even

the development itself with object-oriented programming is made easier. But now a suitable tool has to be found [7].

Parcel is such a module bundling tool to integrate several modules into one file. This makes it possible to send multiple modules to the browser in a single bundle. This library offers some very useful functions, which simplify the implementation considerably. Parcel uses worker processes to enable multicore compilation and has a file system cache to enable very fast rebuilds. Thus Parcel is especially designed for performance in the browser. To further increase performance, Parcel splits the output bundles so that only the required parts are loaded at initial startup. In addition, this library has automatic support for languages such as JavaScript and CSS, so no plugins are required. The program code and the Node.js modules are also transformed automatically. This is very important for this project because it works with Node.js and dependencies. Another interesting feature of Parcel is the automatic update of the modules in the browser, if changes have to be applied by the development. This makes the work process dynamic and faster. The clear and understandable error logging additionally facilitates the implementation [8].

3.2.1 Current Alternatives to Parcel

Parcel is of course not the only module bundling tool, but there are many similar tools. The two well-known tools Webpack and Rollup are roughly outlined in the following section and compared to Parcel.

Webpack Webpack was created to solve asset management problems in a meaningful way and to support the developers in doing so. This is also the reason why non-JavaScript files can be included. While Parcel already provides out-of-the-box support for many languages, Webpack can be adapted and extended by a variety of plugins. In contrast to Parcel, Webpack needs a config file to specify the options for entry, output, plugin etc. more precisely. Parcel does not need one and works automatically. Another difference is that Webpack basically needs a JavaScript file as entry point and does not support an HTML entry point like Parcel. This can be guaranteed by third-party plugins, but is not automatically provided. To create a bundler, all formats have to be converted to JavaScript. This is called transformation. While Webpack uses loaders that have to be defined in the config file, Parcel supports many formats without any additional effort on the part of the developer. Under certain conditions, Webpack can also support the Tree Shaking function, which is responsible for dead code elimination. This would include using the ES2015 format, a sideEffect entry in the package.json file, and adding a minifer that supports dead code removal. Parcel does not support this feature in any way.

To build a development server that facilitates the development and testing of programs, a plugin and the appropriate configurations must be added to Webpack. Parcel has a built-in development server that automatically rebuilds the files each time it changes. Webpack supports the Hot Module Replacement just like Parcel does. Code splitting, to load only the required resources at the beginning and thus improve the performance in the browser, is a very distinctive feature in Webpack. It offers three implementation approaches, between which the user can choose. Once the code can be split manually using the entry configuration, then chunks can be split with the help of a plugin, and the last option is splitting using inline function calls. Parcel supports only one approach to code splitting: zero configuration code splitting. Here the splitting is controlled by the dynamic import function, which loads the modules asynchronously.

In spite of the large number of possibilities Webpack gets from the plugins, it was decided against this tool, because much more time would have to be invested in the configuration, while Parcel does not need any. Parcel already contains all the required features and is ready to go right away, which drives the fast development process even further. This makes it easier and more efficient to use Parcel for development, as it doesn't require a lot of prior knowledge and is easier to use due to its low-configuration usage.

Rollup Rollup is a module bundler for JavaScript, which combines small code snippets into a big overall picture. It uses the new standardized format ES2015 for Code Modules. Like Webpack, Rollup requires a config file, while Rollup also supports relative paths and node polyfills. This requires time before development, while parcel can be started without configuration. To provide the same functionality of Parcel that HTML can also be used as an entry point, a plugin for rollup must be installed. While Parcel performs the transformation for the bundle creation automatically, Rollup requires a suitable plugin to be selected and specified in the config file. Rollup supports Tree Shaking without any prerequisites unlike Webpack. Rollup checks the code to import and exclude the currently unused functions. Unfortunately Parcel does not support this function as mentioned above. To use a development server, an additional plugin has to be installed and configured. In order to have a live reload function, an additional plugin has to be added and configured. It is much more time consuming than using the hands-on support at Parcel. Unfortunately Rollup does not support the Hot Module Replacement like Parcel or Webpack and therefore does not improve the performance in the browser. In the code splitting area there is only experimental code that splits chucks into standard ES modules which are then called by the module loader. Therefore two flags in the config file have to be set differently.

Also after this comparison the decision was made to use Parcel, because Rollup lacks important features, which are relevant for the project and the configuration effort is much higher than with Parcel [9], [10], [11].

3.3 Immutable.js

In the later implementation of the Davis-Putnam algorithm, a known problem will occur when using JavaScript. This is the comparison between sets. The native set compares objects via Object by Object Reference as shown in listing 2. Comparisons with mutable objects are usually very difficult, because when an object changes, the location of the object in a collection also changes. The Immutable.js library enables the creation of immutable objects that cannot be changed once they have been created. It also provides several persistent immutable data structures such as stack, set and range for developers. Thus this library solves the problem of comparisons between objects [12], [13].

```
// plain JavaScript
let set1 = new Set(['1', '2']); // {'1', '2'}
let set2 = new Set(['1', '2']); // {'1', '2'}

// same object twice
var set = new Set([set1, set1]); // {{'1', '2'}}
// different objects with same entries
var set = new Set([set1, set2]); // {{'1', '2'}, {'1', '2'}}
```

Listing 2: Set of sets in plain JavaScript

3.4 Two.js

This library is a two-dimensional drawing API that is used to display and animate SVG files. Two.js also supports canvas and webgl. The main focus of Two.js is clearly on vector graphics, because they were strongly influenced by motion graphics. Therefore, the creation and animation of flat shapes is shortened and easier using Two.js. Two.js is basically based on a scene graph. When an object is drawn or created, the library saves it and remembers it. This allows any operations to be applied to the object after creation. Two.js also includes an animation loop that has been kept very simple to allow automation or combination with other libraries. The last big feature of Two.js is the SVG interpreter, which makes it possible to import SVGs created in popular programs like Adobe Illustrator directly into the Two.js scene. This supports the possibility to create your own SVGs like icons very well [14].

3.5 Seedrandom.js

This library by David Bau provides a seeded random number generator for JavaScript and can be used as plain JavaScript or Node.js module. Seedrandom.js is mainly used to replicate previously created calculations with the same seed. Thus, seeds that can solve a certain problem very quickly or provide an illustrative calculation path for explanation purposes can be reused for the same result at any time [15].

4 Conception

In order to ensure that the program offers the desired functionality at the end of the development process, it is crucial to define the requirements at the beginning and evaluate the project after completion.

4.1 Requirement Analysis

The focus of this project is in the demonstration of the Davis-Putnam algorithm using the *N-Queens Problem*. In order to achieve this in the best possible way, the following requirements must be met.

- The size of N for the *N-Queens Problem* has to be modifiable via a user interface
- With every attempt to solve a certain N-Queen problem, a different starting point should be chosen. As a result different calculation ways are developed and various solutions are shown. However, it must be guaranteed that there is always the possibility to show replicable results.
- The process to solve the *N-Queens Problem* must be shown visually on a chessboard in individual steps on the one hand and reproduced as individual calculation steps on the other hand. The user must be able to choose between Macro or Micro steps. Depending on which mode was selected, the size of the intermediate steps changes.
- The simulation must be pausable.

At the end of the implementation phase, the program is measured against these requirements and declared as successful or unsuccessful, as the case may be.

4.2 Layout Design

In order for users to view and understand the animation of the Davis-Putnam algorithm in the best possible way, it is important to provide a clear and concise user interface. The interactions and settings must be self-explanatory and understandable. Therefore, different ideas for a possible layout were collected in advance and the following design was developed.

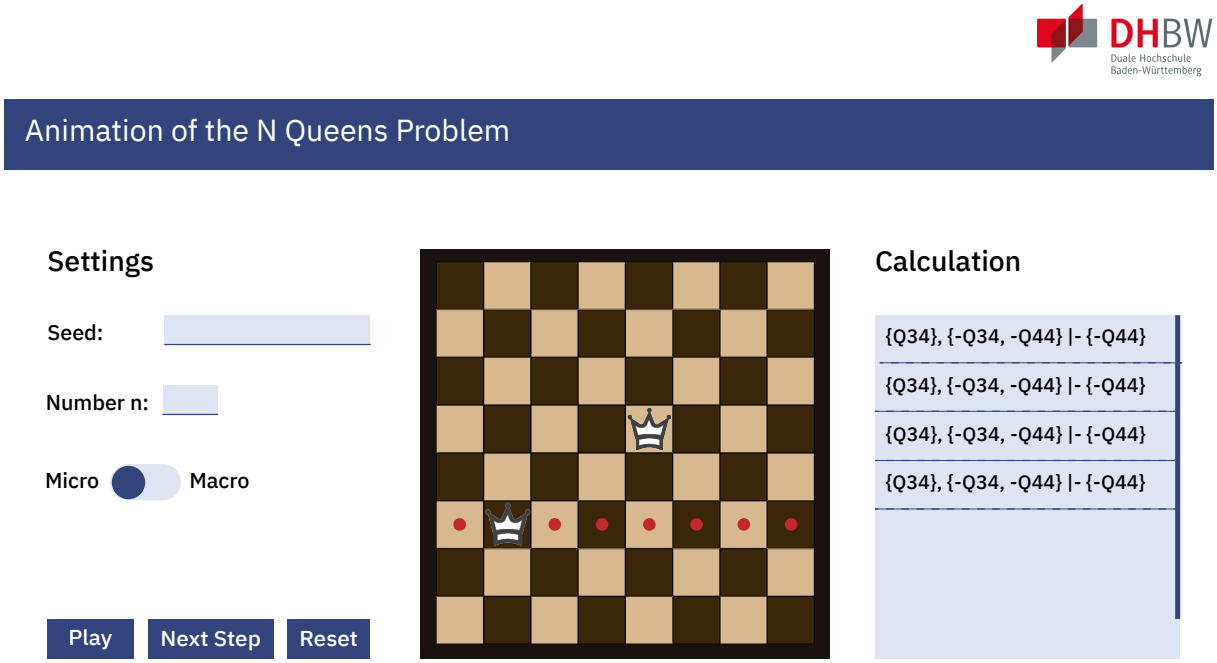

Figure 4.1: Final Design Sketch for User Interface

The header serves as an eye-catcher so that the topic of the website immediately stands out clearly and comprehensibly.

Below, the user interface is visually divided into three parts. The chessboard is in the middle, because the main animation takes place here and should be the center. It immediately captures the view of the users and directs them to the main event.

To the left are the settings for the animation and execution of the algorithm. The result is a natural reading flow from left to right, from the settings via the chessboard to the calculations. The user is navigated through the system in an intuitive and self-explanatory way, without any conspicuous control elements. The settings contain only the most important elements with which the animation can be adjusted. Thus it does not appear overloaded anywhere, but clear and structured. Below are the buttons for controlling the animation. Depending on the situation, the functionality is adapted to it. In this design the settings take a fixed place on the surface, but there was also the idea to convert the settings into a sidemenu. However, the fact that the user has to interact permanently with the control buttons and that the fold-out menus are only suitable for one-time interactions, such as navigating through a page, spoke against this.

On the right side of the user interface is the calculation, which shows the mathematical background process in individual steps. The individual steps are displayed in a scrollable box. Whether the top element always remains on top or the newer elements are placed on top is not defined in the design itself.

It is important that all components are coherent with each other. This means that positioning, distance and alignment are consistent. This is the only way to create a coherent overall picture. The colours have also been chosen in such a way that they appear calming and in a uniform tone, but with different saturations. Thus the surface appears harmonious and not overloaded. The brown of the chessboard fits very well to the blue tone in the surroundings and stands out as the only other colour from the mass. However, this colour scheme was only used for demonstration purposes and orientation. It can be adapted in the course of implementation and replaced by a more suitable set. A well-structured and clear layout has been created, which is easy to understand and use for the users.

5 Implementation

As mentioned before in the chapter 3 we will be using Node.js with Parcel as a module bundler. This way we can use Node.js modules and build an HTML page that does not require a Node.js server running in the background, making it portable and executable without any prerequisites except for a browser. The implementation in JavaScript will be an object oriented approach using the class syntax introduced in ECMAScript 2015, which will make it more pleasing for the eye and less confusing [16].

Since the Davis-Putnam algorithm can have tasks with a high computational effort, we decided to use separate threads to compute and animate. So we can display all the changes to the problem as it is being satisfied and prevent the GUI from freezing on long lasting computations. As previously explained in chapter 3.1 this is possible with a so-called worker in JavaScript. It is important to mention here, that because the worker and main script communicate with each other via messages, we need to make sure that while computing no unwanted messages are send. Such messages could be multiple computing request being send almost simultaneously to the worker, because the worker will buffer send messages and respond to them after finishing all previous tasks. This could lead to problems if for example during a long lasting computation task additional computations are requested, then no other requests could be handled before all those computation tasks have been finished nor could those be interrupted. Our approach here will be to tell the main script when a computational task started and ended. This way certain UI elements can be locked while the worker computes and unlocked when the worker finishes the given task.

Next we need to take a look at the mathematical part of the script. Because JavaScript uses references when comparing objects, we will use Immutable.js as discussed in chapter 3.3 to counter this problem when working with sets. Since we will have a set of clauses for the algorithm to satisfy with those clauses being sets of literals, we would need to implement a deep compare ourselves if we were to use native JavaScript. Now that we know that Immutable.js will solve the set comparison problem, the literals still need to be implemented somehow. We decided to use simple strings for the literals with the restriction of the first character not being allowed to be an “!”, as we will use that character to indicate that the literal is negated. The choice to use strings is based on the same reason for using Immutable.js for the sets. Immutable.js has immutable structures that are compared by value, but for other objects Immutable.js compares them just like native JavaScript by their object references. This means if we were to use custom classes for literals we would

need to implement a deep compare anyway, but strings on the other hand are compared by length and character order by default. Resulting in less effort when implementing the mathematical part of the algorithm [17], [18], [19].

So for example the formula $(p \vee \neg q) \wedge (s \vee r)$ would be implemented as the following listing 3 with Set being the Set class of Immutable.js and not the native JavaScript one.

```
// (p or not q) and (s or r) as a set of clauses in JavaScript
let set_of_clauses = new Set(new Set('p', '!q'), new Set('s', 'r'));
```

Listing 3: Example for a set of clauses in JavaScript

Now that it is clear how a set of clauses will be implemented in JavaScript and the decision to use two threads has been made, we can start getting into the classes and their tasks.

5.1 Util Object

Because some functions like negating a literal will be used in multiple parts of the script, we will use an object to host these functions. This object will be called the Util Object. By doing so it can be imported when those functions are needed and prevent redundant code, making it easy to maintain or change.

It contains the function to negate a literal and functions for formatting sets and literals to HTML strings. It can be expanded further in future, but for now these will do. Because we are using strings for literals we can negate them by adding a "!" and applying a simple RegEx operation afterwards as shown in the following listing 4.

```
Util.negateLiteral = (literal) => return ('!' + literal).replace(/^!!/, '');
Util.negateLiteral('p');  // result: !p
Util.negateLiteral('!p'); // result: p
```

Listing 4: Example for negating a literal in JavaScript

Formatting sets and literals to HTML, using the span element and adding classes to it, will allow us to color code them in the GUI. This will make it easier for the user to interpret the visuals of the animation.

5.2 Implementing the Algorithm of Davis-Putnam

The original *Davis-Putnam algorithm* is recursive and so are most of its implementations. If you were to calculate everything till you find a possible solution or the response that the

given problem can not be satisfied, then a recursive implementation is not an issue. For our purposes, which is the visualization of the algorithm as it solves the given problem step by step, we prefer to use an iterative implementation. If we use a recursive implementation we would need to calculate everything upfront, which is not the case for a iterative implementation. These can be paused at any point and because of that we can calculate the step when we need it to be calculated. This means only the creation of the problem that will be satisfied by the algorithm will affect the initial loading time.

One of the challenges here is the double recursive call of the algorithm and splitting it into single unit cuts per iteration for the micro steps. The basic idea will be a class that holds the current state of the problem with a step function that does either a macro or micro step further satisfying the problem till it is satisfied or not satisfiable. One macro step will include all micro steps till the algorithm guesses the next literal, with micro steps being single unit cuts. To be able to do this the class also needs to know what has been done previously to continue from that point. With that in mind the following points will be able to describe the state of the algorithm, so that it can continue from where it stopped.

1. The set of clauses describing the current state of the problem
2. The literals it has guessed previously
3. The unit clauses it has already used for further satisfaction
4. The unit clause or literal that is currently in use for unit cuts

The last point is important for the micro steps, because we need to do all possible unit cuts before using the next unit clause. Because it is possible to represent the state of the algorithm, we can also use this to simulate the double recursive call. By pushing the current state with the negation of the guessed literal to the stack and adding the guessed literal to the current state for further satisfaction, we can pop a state from the stack to continue from that point if the current state is not satisfiable.

Because we will use an iterative implementation, we need to think about everything that needs to be done in one step of the iteration. Since we will have macro and micro steps, we will use two loops one for the macro steps and one for the micro steps, so that one macro step will execute all micro steps that it includes. Lets say we have a function step that receives a number of steps n so that it will do n micro or macro steps on the state. Then we need to reduce the step count when a guessed literal L has been added to the state, in case of macro steps, or a unit cut has been calculated, for micro steps. The function step would look like the pseudo code of listing 5. Doing all micro steps at the start of a macro step results in checking if it is satisfied. If it is not satisfiable, it is followed by a backtrack and the guessing of a new literal.

5.2.1 Davis-Putnam Class

As previously explained we need a stack of states and a state to provide an iterative implementation of the *Davis-Putnam algorithm*. For this purpose we create a Davis-Putnam class. Its primary features are that it contains a state, a stack of states and a function to execute a macro or micro step on the state. Because the algorithm is supposed to be animated we have to pass the operations on the set of clauses and the unit clauses of the set of clauses to represent the state. This can be done with event listeners, that are called on the important operations of the algorithm. Since there are a number of events we decided to use a class that contains functions to be called when those events happen. These events are as follows:

1. set of clauses satisfied
2. set of clauses not satisfiable
3. backtrack
4. choose a literal
5. subsume
6. unit cut

Each of those events calls will receive the important information needed to describe the operation and state of the set of clauses. They all receive the literals of the unit clauses, so we can represent the state at those points. The first three do not need any additional information, but the later do. When a literal is chosen said literal will also be handed to the function for the event. The function for subsume will receive all removed clauses and the function for unit cut receives the used literal and the clause before and after the operation. This way all operations during one step can affect the animation.

Since the animation is for educational purposes, it is also important to be able to replicate the results. Therefor we will be using Seedrandom.js as mentioned in chapter 3.5 at this point. When choosing a random element in this class we will use our own random number generator that we initialize with a seed. This way each seed has its unique repeatable calculation. It is important to mention here that this is also only possible because of the way the Set class from Immutable.js works. “When iterating a Set, the entries will be (value, value) pairs. Iteration order of a Set is undefined, however is stable. Multiple iterations of the same Set will iterate in the same order [20].”

5.2.2 Davis-Putnam Consumer Class

This class is for handling the events of the Davis-Putnam class. It contains the following functions:

1. onSatisfied
2. onNotSatisfiable
3. onBacktrack
4. onChoose
5. onSubsume
6. onUnitCut

Those functions are basically placeholders, since the class is supposed to be extended by a subclass. The subclass should alter these functions to their needs. Other than that this class has no further purpose.

5.2.3 Davis-Putnam Worker Class

As said before we will have a worker for the calculations. This class initializes a worker for the main script to do all the calculations. It extends the Davis-Putnam Consumer from the previous chapter 5.2.2. It will create its own Davis-Putnam instance and add itself as a consumer, so that all events from said instance will call the functions of this class. When these functions are called the worker will send messages to the main script with all the information. In addition to those messages it will also tell the main script when it starts or stops calculating, because of the potential problem named previously in chapter 5. These will be all outgoing messages. As for the incoming messages from the main script. We will accept requests for a new set of clauses, a seed for the Davis-Putnam class, switching between micro and macro steps and to calculate the next step.

Because the worker and main script communicate via messages only, there is only one listener for all those events. To be able to distinguish between the different requests we will define a default message structure as shown following listing 6.

The “cmd” value will indicate what kind of request it is and the “options” object will contain all the information needed to fulfill that request. This way the listener can be implemented with a simple switch-case to handle the requests as shown in following listing 7.

With the components of the calculation thread finished, we are only missing those of the animation thread.

5.3 Implementing N-Qeens Problem

To be able to solve a problem with the *Davis-Putnam algorithm*, a propositional formula needs to be defined in JavaScript. Therefore we will create a function to return for a given number n a set of clauses that express the *N-Queens Problem* for said n. The queens will be represented by a literal of the form "row,col" and the negated literal will mean that no queen can be placed there. Lets say for example we have $\texttt{row} := 3 \wedge \texttt{col} := 4$ then the literal for the queen would be as follows "3,4" and the negation would be "!3,4". This makes it easy to transform the string back into coordinates for the chessboard.

5.3.1 Queens Clauses Function

This function returns a set of clauses for the *N-Queens Problem*, so that for a given n the problem will be to place n queens on a $n \times n$ board. To create this set of clauses following functions are used to help:

atMostOne Returns a set of clauses that allows only one of the given literals to be true, for a given set of literals.

atMostOneInRow Returns a set of clauses where only one of the row can be true, for a given n and *row*.

oneInColumn Returns a set of clauses where at least one in the column is true, for a given n and *column*.

atMostOneInUpperDiagonal Returns a set of clauses where only one in the upper diagonal can be true, for a given n and helper variable k.

atMostOneInLowerDiagonal Returns a set of clauses where only one in the lower diagonal can be true, for a given n and helper variable k.

5.3.2 Chessboard Class

To visualize the *N-Queens Problem*, we will use Two.js as mentioned before in chapter 3.4. The Chessboard class will receive a number n and create a visual representation of a $n \times n$ chessboard. Since the state of the set of clauses will be represented by the literals of the unit clauses, we will need two different indicators per tile of the board. We decided to represent the queen by a green dot and a negative literal by a red cross.

The chessboard will have four layers the first being the board itself, the second for the $n \times n$ tiles of the board, the third for the queens and the last for the crosses. This way we can initialize the board with all needed Two.js objects and add or remove them from their specific layers. To only add or remove changes of one state to the other, the chessboard will remember the state to be able to compare it with the new state. This will result in a smoother animation and less visual changes.

As for scaling of the chessboard, we will scale the Two.js scene to fit the parent HTML container with a listener that gets called when the window resizes.

5.4 Implementing User Interaction

Now that we have the *Davis-Putnam algorithm* and *N-Queens Problem* set up in JavaScript, we need to enable the user to interact with it. In chapter 4 we talked about the design. Said design will be used as a reference for the GUI of the animation. Therefor we need an HTML, CSS and JavaScript file for the webpage.

As for the interaction of the User with the animation, we will enable the user to do the following:

1. Change the seed for the Davis-Putnam class
2. Change the number n for the *N-Queens Problem* affecting also the visualization
3. Toggle between micro and macro steps
4. Request the next step to be animated
5. Request to automatically animate all steps until requested to stop or no further steps can be done
6. Reset the whole animation to run it again from the start
7. Display previous calculated states of the shown steps

For all points except the last one we got corresponding elements for interaction incorporated into the design. To enable the user to display previous states, we decided to make the elements of the calculation history clickable.

5.4.1 HTML Document

The HTML file contains all important elements needed for the GUI with their unique ids to access them via JavaScript. No additional GUI elements will be added with JavaScript except for populating the calculation log. For changing GUI elements, we will be using the "checkbox hack". This way we do not need to change the style of the GUI with JavaScript and additional listeners, focusing the JavaScript part only on the interaction with the worker and visual representation by the chessboard [21].

5.4.2 CSS Document

For the layout we decided to use a grid based design, to be able to make it somewhat responsive. We added only one break-point for the smallest size we want the GUI to have, so that the chessboard has an acceptable minimal size. For larger screens it will scale up accordingly.

5.4.3 Frame Class

To initialize the animation and all its needed components, we created a Frame class. This class contains all the functions for the user interaction, the worker for the calculations and the chessboard object for visualization. It creates all the listeners for the UI elements and handles the communication with the worker. To synchronize the animation with the worker the User Interface elements that should not be triggered will be disabled during the calculation process of the worker and enabled at the end of said calculation. This way we prevent unwanted behavior of the worker. Whenever the worker sends a message containing an important step of the calculation a new entry will be created for the calculation history. New entries will be added before the previous entries, so that the top entry is the latest and the bottom one the first calculation step.

Each of the named possible user interactions receives their own listener, that handles the requested task. On top of those we need one more for the messages of the worker, because we will have a two way communication. This listener is very similar to the one for the worker as previously shown in listing 7, only that it will handle all outgoing messages of the worker.

5.5 Result

The result of the implementation can be viewed and interacted with at GitHub Pages of koenloogman. The animation is optimized for the latest version of Google Chrome or Firefox and maybe works on other browsers as well, but we recommend said two browsers for the best experience. If interested in the full source code, it can be seen at GitHub of koenloogman.

```
/* do step(s) on a state that contains a set of clauses S */
function step( number of steps n ) return
    repeat /* macro step loop */
        repeat /* micro step loop for subsume and unit cuts */
            /* select a new literal and subsume till unit cuts can be done with said literal */
            repeat
                choose a unit clause U that has not been used yet
                add U to used unit clauses of the state
                choose and remember literal L from U
                /* unit subsumption */
                delete from S every clause containing L
            until all unit clauses have been used or clauses in S contain L̄
            /* check if cuts can be done before trying to do a unit cut */
            if clauses in S contain L̄ then
            /* single unit resolution */
                choose a clause C that contains L̄
                delete L̄ from C
                if it is a micro step then
                    n− = 1
                fi
            fi
        until all unit clauses have been used or n = 0
        /* check if satisfied */
        if S contains only unit clauses then
            /* is satisfied */
            continue
        else if n = 0 then
            continue
        fi
        n− = 1
        /* check if current state is not satisfiable */
        if a clause becomes {} in S then
            if stack is empty then
                /* is not satisfiable */
                set S = { {} }
                continue
            fi
            pop state from the stack
            continue
        fi
        choose and remember a literal L occurring in S
        push state with literal L̄ to the stack
        add literal L to the current state
    until no further steps can be done or n = 0
    return
end function
```

Listing 5: Iterative step for *Davis-Putnam algorithm*

```
// message example
let message = {
    'cmd': command_name,
    'options': {
        'option_1': value_1,
        'option_2': value_2
    }
}
```

Listing 6: Worker message example

```
// listen to events
self.addEventListener('message', event => {
    let cmd = event.data.cmd;
    let options = event.data.options;

    // handle commands
    switch(cmd) {
        case 'seed':
            // handle seed change
            break;
        case 'clauses':
            // handle clauses change
            break;
        case 'micro':
            // handle micro/macro change
            break;
        case 'step':
            // handle step
            break;
        default:
            // undefined command
    }
});
```

Listing 7: Event listener of the Davis-Putnam Worker class

6 Prospect

6.1 Evaluation

After the development phase, the time has come to measure and evaluate the program against the requirements defined in chapter 4. These criteria were specified according to the desired functions and objectives of the end user and must all be met for the program to be considered successful. In the following, the previously defined criteria are examined individually and either assessed as fulfilled or not.

Modifiable Number N The second input field on the user interface allows the user to set a desired number N for the *N-Queens Problem*. Thus, this requirement is considered fulfilled.

Possibility to show replicable results With the help of the seed, which can be entered or changed personally in the first input field, it is possible to replicate certain invoice sequences again and again by selecting the same N and the same seed. This can be seen quite well in the following example.

This example shows the calculation path of the algorithm with the seed “Exmatrikulator” for the 4-Queens Problem. Whenever this seed is used for this particular problem, the exact calculation path is followed. At the chessboard, the solution can be seen in the following picture.

```
Choose ‘2,1’
{ ‘2,1’ }, { ‘!3,2’, ‘!2,1’ } ⊢ { ‘!3,2’ }
{ ‘2,1’ }, { ‘!2,1’, ‘!1,2’ } ⊢ { ‘!1,2’ }
{ ‘2,1’ }, { ‘!4,3’, ‘!2,1’ } ⊢ { ‘!4,3’ }
{ ‘2,1’ }, { ‘!2,4’, ‘!2,1’ } ⊢ { ‘!2,4’ }
{ ‘2,1’ }, { ‘!2,2’, ‘!2,1’ } ⊢ { ‘!2,2’ }
{ ‘2,1’ }, { ‘!2,3’, ‘!2,1’ } ⊢ { ‘!2,3’ }
{ ‘!4,3’ }, { ‘1,3’, ‘2,3’, ‘3,3’, ‘4,3’ } ⊢ { ‘1,3’, ‘2,3’, ‘3,3’ }
{ ‘!2,3’ }, { ‘1,3’, ‘2,3’, ‘3,3’ } ⊢ { ‘1,3’, ‘3,3’ }
{ ‘!1,2’ }, { ‘1,2’, ‘2,2’, ‘3,2’, ‘4,2’ } ⊢ { ‘4,2’, ‘2,2’, ‘3,2’ }
{ ‘!2,2’ }, { ‘4,2’, ‘2,2’, ‘3,2’ } ⊢ { ‘4,2’, ‘3,2’ }
{ ‘!2,4’ }, { ‘1,4’, ‘2,4’, ‘3,4’, ‘4,4’ } ⊢ { ‘1,4’, ‘4,4’, ‘3,4’ }
{ ‘!3,2’ }, { ‘4,2’, ‘3,2’ } ⊢ { ‘4,2’ }
{ ‘4,2’ }, { ‘!4,2’, ‘!3,3’ } ⊢ { ‘!3,3’ }
{ ‘4,2’ }, { ‘!4,4’, ‘!4,2’ } ⊢ { ‘!4,4’ }
{ ‘4,2’ }, { ‘!4,2’, ‘!4,1’ } ⊢ { ‘!4,1’ }
{ ‘4,2’ }, { ‘!4,2’, ‘!3,1’ } ⊢ { ‘!3,1’ }
{ ‘!3,3’ }, { ‘1,3’, ‘3,3’ } ⊢ { ‘1,3’ }
{ ‘!4,4’ }, { ‘1,4’, ‘4,4’, ‘3,4’ } ⊢ { ‘1,4’, ‘3,4’ }
{ ‘1,3’ }, { ‘!3,1’, ‘!1,3’ } ⊢ { ‘!3,1’ }
{ ‘1,3’ }, { ‘!1,4’, ‘!1,3’ } ⊢ { ‘!1,4’ }
{ ‘1,3’ }, { ‘!1,3’, ‘!1,1’ } ⊢ { ‘!1,1’ }
{ ‘!1,4’ }, { ‘1,4’, ‘3,4’ } ⊢ { ‘3,4’ }
Problem satisfied
```

Listing 8: Calculation result using the seed “Exmatrikulator” for the 4-Queens Problem

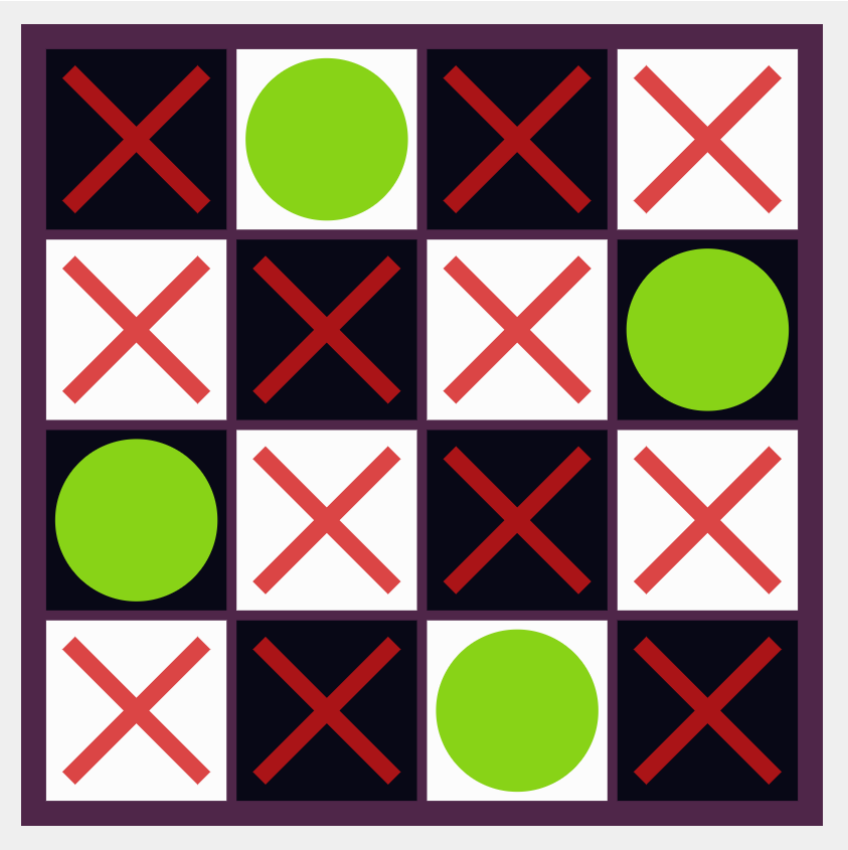

Figure 6.1: Final Solution for 4-Queens Problem with Seed "Exmatrikulator"

Thus it is possible to replicate a certain result as often as possible and at any time. Thus the requirement is considered fulfilled.

Visualization of current progress The criterion here was to display the current solution process as a visual representation on the chessboard and as calculation steps. On the user interface, the animation is displayed visually in the middle of the chessboard and the calculations are shown in a box to the right. It should also be mentioned that this can either be done automatically or step by step with a single push of a button. The step size can also be adjusted. You can choose between Micro and Macro. Thus this requirement is also fulfilled.

Pausable Progress The requirement was that the simulation could be paused. If the user pressed the "Play" button, the simulation starts and at the same time the start button turns into a "Stop" button. This button can be used to stop the animation by pressing it. Please note, however, that the simulation only stops after the current calculation step has been completed. After this has been stopped, the button changes back to "Play" and the animation can be continued at the corresponding position.

6.2 Conclusion

As shown in the previous section, this program meets all previously defined requirements. The program is considered successful and the project can now be completed satisfactorily. In summary, it was a very exciting project with no significant problems. Again and again during the implementation phase, the requirements had to be re-examined and decisions had to be made with the help of these decisions. The program had to be easy to use and understand at all times. Both usability and user experience played a role in all areas of the development process.

The use of Javascript made web development a lot easier, but there were also some difficulties. But these were solved by helpful libraries and technologies like web workers.

6.3 Outlook

There are always areas of a program that can be improved in the future for example the performance. But because the program is only used for small numbers of the *N-Queens Problem*, the speed of the calculations for the larger ones is not the focus for future improvements.

Another idea for future development would be the use of this program for further logical problems and not only the *N-Queens Problem*. By using the web worker this would be possible. Therefore the Davis-Putnam algorithm is universally applicable in this program.

Bibliography

[1] H. Zhang and M. E. Stickel, *Implementing the davis–putnam method*, https://www.math.ucdavis.edu/~deloera/TEACHING/MATH165/davisputnam.pdf, Accessed on 2018-10-23, Oct. 2000.

[2] K. Stroetman, *Theoretical Computer Science I : Logic*. GitHub, Feb. 16, 2019. [Online]. Available: https://github.com/karlstroetmann/Logik/blob/master/Lecture-Notes-Python/logic.pdf (visited on 03/06/2019).

[3] The Oxford Math Center Authors. (May 21, 2019). The n-queens problem & backtracking, [Online]. Available: http://www.oxfordmathcenter.com/drupal7/node/627.

[4] Geeks for Geeks. (2019). N queen problem | backtracking-3, [Online]. Available: https://www.geeksforgeeks.org/n-queen-problem-backtracking-3/.

[5] M. Peng. (Sep. 15, 2017). Multithreading javascript, A look into web workers, [Online]. Available: https://medium.com/techtrument/multithreading-javascript-46156179cf9a (visited on 04/05/2019).

[6] E. Bidelman. (Jul. 26, 2010). Web worker-grundlagen, Das problem: Nebenläufigkeit von javascript, [Online]. Available: https://www.html5rocks.com/de/tutorials/workers/basics/ (visited on 04/05/2019).

[7] P. Kasireddy. (Feb. 5, 2016). Javascript modules part 2: Module bundling, [Online]. Available: https://medium.freecodecamp.org/javascript-modules-part-2-module-bundling-5020383cf306 (visited on 04/05/2019).

[8] D. Govett. (2019). Parcel, [Online]. Available: https://parceljs.org/ (visited on 04/05/2019).

[9] G. Bhatia. (Jun. 12, 2018). Comparing bundlers: Webpack, rollup & parcel, [Online]. Available: https://medium.com/js-imaginea/comparing-bundlers-webpack-rollup-parcel-f8f5dc609cfd (visited on 04/05/2019).

[10] Webpack Authors. (Apr. 1, 2019). Concepts, [Online]. Available: https://webpack.js.org/concepts (visited on 04/05/2019).

[11] rollupJs-Authors. (2019). Rollup.js, Introduction, [Online]. Available: https://rollupjs.org/guide/en (visited on 04/05/2019).

[12] Immutable.js Authors. (2019). Immutable collections for javascript, [Online]. Available: https://immutable-js.github.io/immutable-js/ (visited on 04/05/2019).

[13] A. Rauschmayer. (Jan. 5, 2015). Ecmascript 6: Maps and sets, [Online]. Available: http://2ality.com/2015/01/es6-maps-sets.html#why-cant-i-configure-how-maps-and-sets-compare-keys-and-values (visited on 04/05/2019).

[14] TwoJs-Authors. (2019). Two.js, Introduction, [Online]. Available: https://two.js.org/#introduction (visited on 04/05/2019).

[15] D. Bau. (Mar. 4, 2019). Seedrandom.js, [Online]. Available: https://github.com/davidbau/seedrandom (visited on 05/04/2019).

[16] Mozilla Developers. (Apr. 16, 2019). Defining classes, [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes.

[17] Mozilla Firefox Developers. (Mar. 18, 2019). Equality operators, [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators#Equality_operators.

[18] Mozilla Developers. (Apr. 6, 2019). Object.is(), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/is.

[19] Immutable Developers. (2019). Immutable js object.is(), [Online]. Available: https://immutable-js.github.io/immutable-js/docs/#/is.

[20] Immutable.js Developers. (2019). Immutable set, [Online]. Available: https://immutable-js.github.io/immutable-js/docs/#/Set.

[21] C. Coyier. (Feb. 14, 2012). Stuff you can do with the “checkbox hack”, [Online]. Available: https://css-tricks.com/the-checkbox-hack/.