RPTU Rheinland-Pfälzische Technische Universität Kaiserslautern Landau

A Domain-specific Language for Recreational Mathematics

Bachelor's Thesis

by

Aurelijus Kadzys

December 11, 2023

University of Kaiserslautern-Landau
Department of Computer Science
67663 Kaiserslautern
Germany

Examiner: Prof. Dr. Ralf Hinze
Michael Sherif Kamel Youssef

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit mit dem Thema „A Domain-specific Language for Recreational Mathematics“ selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit — einschließlich Tabellen und Abbildungen —, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Kaiserslautern, den 11.12.2023

Aurelijus Kadzys

Abstract

This thesis introduces a way of solving recreational problems in an easy to understand and intuitive manner. The problems that are looked at in particular are exact cover. That means that they are decision, as well as constraint satisfaction problems. The already existing Algorithm X by Donald Knuth is created to solve these specific problems. However, for a problem to be solved, the constraints need to be identified and programmed into a procedure, so that it can give a format that Algorithm X can use to solve.

Instead a domain-specific language is built that is able to create the format without needing to code anything. The domain-specific language would be able to be used by anybody with some expertise in exact cover problems. This is due to only needing to understand the problems constraints and how the exact covering works. Not only would it make it easy to solve the specific kinds of problems, it would also a good tool to create new problems. This is a consequence of the possible straightforward application, and availability of constraints, making it easy to experiment with different constraints and board-states.

However, the domain-specific language presented in this thesis is still quite narrow. As such the limitations are explored and example extensions are presented to show the future potential of the language.

Contents

1 Introduction

Most people have experience with puzzles such as Crosswords, Sudoku, and Rubik's Cubes. These recreational activities are fun to solve but sometimes there is a need for an algorithm. Sometimes to check for solutions, sometimes to create a puzzle themselves.

The interesting part is, that the aforementioned puzzles have a lot in common. Namely, they all are so called **Exact Cover Problems**. It means that all the tiles can only be covered exactly once, even if one has enough options to cover it many times. The problem is to find a bundle of pieces that:

A. all fit together (no overlaps)

B. cover the defined surface area

As an example, Sudoku sometimes has multiple possibilities to cover one specific cell. But (**A.**) one cannot have a solution in Sudoku with overlapping pieces. In this case multiple solutions would be valid. Another rule (**B.**) states, that all cells in Sudoku have to be covered. This is, however, not a necessity for an exact cover problem, because sometimes the defined surface area might be to cover each row or column exactly once, like in the puzzle **Queens**.

Exact cover problems are not only restricted to recreational activities. Here is an example from the real world:
A team is put together for a project. This team should have competencies in the areas of architecture (A), building physics (B), chemistry (C), data processing (D), electrical engineering (E), and financing (F). A team member can have several skills. However, no competence should be represented more than once, as that would be a waste of resources.
The following five people are available: Anna has competencies in architecture and building physics, Boris in architecture, building physics and chemistry, Charlotte in chemistry and electrical engineering, Dennis in data processing and financing, Emma in electrical engineering and financing.

If we would take Boris, then Charlotte would drop out. To cover D, E, and F, we are forced to use both Dennis, and Emma. However, as they both overlap on F, we are only able to take one of them. So either D, or E would be left uncovered. This means Boris cannot be taken. That forces us to use Anna, and Charlotte. Taking Charlotte makes Emma drop out, and leaves only Dennis as the only other option. Because Dennis takes care of the last two competencies,

Competencies						
Person	A	B	C	D	E	F
Anna	X	X				
Boris	X	X	X			
Charlotte			X		X	
Dennis				X		X
Emma					X	X

Table 1.1: *Example of exact cover problem*

we have a solution to the problem, as all the competencies are covered exactly once.

$$\begin{aligned} X &:= \{A, B, C, D, E, F\} \\ S &:= \{\{A, B\}, \{A, B, C\}, \{C, E\}, \{D, F\}, \{E, F\}\} \\ \implies U &:= \{\{A, B\}, \{C, E\}, \{D, F\}\} \end{aligned}$$

Puzzles that are difficult for humans are likely to be also difficult for computers. This applies in this scenario, as exact cover is known as one of Karp's 21 NP-complete problems[1]. Then a problem is classified as NP, it means that it has a running time of **nondetermistic polynomial time**. Therefore, if we have a nondeterministic computation, the problem would be solvable in polynomial time. However, one of the open questions in computer science is NP $\overset{?}{=}$ P (polynomial). At the time of publication we do not have an answer to this question. Thus, we are only able to create exponential time algorithms for NP-complete problems.
As such, we need an efficient algorithm to solve exact cover, else the algorithm would not be applicable in practice. Donald Knuth is one of the people who designed such an algorithm, and his work is currently the state of the art for solving exact cover problems.

Because his algorithm needs a specific formatting to work, somebody has either to write the whole format per hand, or write a program that creates such a format. Both solutions are tedious, and creating an algorithm that does it for one, needs the person to have coding experience. The goal for this thesis is to create a domain-specific language to make the process of creating a format as easy as possible while needing only a little knowledge of exact cover. Furthermore, we are going to use Haskell as the host language for the program, due to having ample of tools to creating one with ease.

[1]Karp has shown in 1972 for 21 different computational problems that they are NP-Complete, exact cover being one of them

2 Background

2.1 Exact Cover

Exact cover is a collection of subsets $\mathcal{S}$ of a set X such that each element in X is contained in *exactly one* subset in $\mathcal{S}$. In other words, each element is covered by exactly one subset.
An exact cover problem is a kind of constraint satisfaction problem[1]. The elements of the subsets $\mathcal{S}$ are considered choices/**options**, while the elements of set X represent constraints, also called **items**.

Sometimes constraints are needed which do not need to be covered but also should not be covered more than once. These *at most once* constraints are called **secondary items**, while the *exactly once* constraints are called **primary items**.
While it is possible to convert the secondary constraints to primary constraints with slack variables[2], it is better to adjust ones algorithm to them, thus saving mermory and computation time.

As an example let us take a look at the generalized exact cover problem **n Queens**. The goal of the puzzle is to place n queens on an $n \times n$ board without the queens attacking one another.
What are the constraints for this puzzle?
As queens are able to move vertically and horizontally, it means that on an $n \times n$ board n queens can only be placed on seperate rows and columns. These are our **primary constraints**. Queens are also able move on the upward diagonal, and on the downward diagonal. However, as there are more diagonals than queens, namely $2 \cdot n - 1$, not all of the diagonals can be covered. Even so, no diagonal can be represented more than once. This makes the diagonals **secondary constraints**.

[1] mathematical questions defined as a set of objects whose state must satisfy a number of constraints or limitations
[2] Slack variables can change a secondary item to a primary one by adding a 0 or 1 depending on how often the secondary item was selected

The constraints can be summarized like this:

$$\text{Row constraint: } \sum_{i=1}^{n} x_{ij} = 1 \text{ for } 1 \leq j \leq n \tag{2.1}$$

$$\text{Column constraint: } \sum_{j=1}^{n} x_{ij} = 1 \text{ for } 1 \leq i \leq n \tag{2.2}$$

$$\text{Upward constraint: } \sum \{x_{ij} \mid 1 \leq i, j \leq n, i + j = s\} \leq 1 \text{ for } 1 < s \leq 2n \tag{2.3}$$

$$\text{Downward constraint: } \sum \{x_{ij} \mid 1 \leq i, j \leq n, i - j = d\} \leq 1 \text{ for } -n < d \leq n \tag{2.4}$$

With the constraints defined, we can define our items. r_i is chosen as row i (i being defined in 2.1), c_j as column j (2.2), a_s as the upward diagonal s (2.3), and b_d as the downward diagonal d (2.4). The option 'r_i c_j a_{i+j} b_{i-j}' represents a queen placed on cell X_{ij}.
For example, when n = 4 the placement options are:

$r_1\ c_1\ a_2\ b_0$	$r_2\ c_1\ a_3\ b_1$	$r_3\ c_1\ a_4\ b_2$	$r_4\ c_1\ a_5\ b_3$
$r_1\ c_2\ a_3\ b_{-1}$	$r_2\ c_2\ a_4\ b_0$	$r_3\ c_2\ a_5\ b_1$	$r_4\ c_2\ a_6\ b_2$
$r_1\ c_3\ a_4\ b_{-2}$	$r_2\ c_3\ a_5\ b_{-1}$	$r_3\ c_3\ a_6\ b_0$	$r_4\ c_3\ a_7\ b_1$
$r_1\ c_4\ a_5\ b_{-3}$	$r_2\ c_4\ a_6\ b_{-2}$	$r_3\ c_4\ a_7\ b_{-1}$	$r_4\ c_4\ a_8\ b_0$

Table 2.1: *Options for 4 Queens problem*

The goal is to select the options where each row r_i and column c_j are represented exactly once, and each diagonal a_s and b_d is represented at most once. In this case we have two solutions:

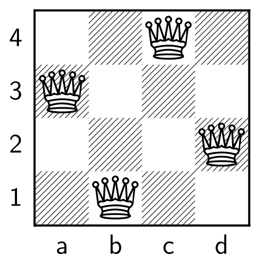
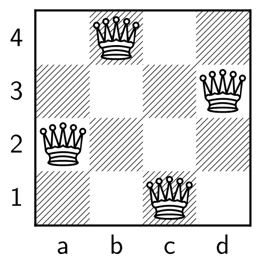
Solution A	Solution B
	
$r_1\ c_2\ a_3\ b_{-1}$	$r_1\ c_3\ a_4\ b_{-2}$
$r_2\ c_4\ a_6\ b_{-2}$	$r_2\ c_1\ a_3\ b_1$
$r_3\ c_1\ a_4\ b_2$	$r_3\ c_4\ a_7\ b_{-1}$
$r_4\ c_3\ a_7\ b_1$	$r_4\ c_2\ a_6\ b_2$

Table 2.2: *Solutions for 4 Queens problem*

Solving small problems like this per hand is not so hard, but it gets exponentially more difficult as the constraints and options increase. That is why an algorithm that can do this efficiently would be greatly appreciated. Donald Knuth's **Dancing Links** program is able to do just that. It sorts through all the options quickly and finds all the possible solutions. This will be covered in depth in Chapter 3.1.

2.2 Domain-Specific Language

A domain-specific language (**DSL**) is a computer language specialized to a particular application domain. They are created to solve problems in that particular domain, and are not intended to be able to solve problems outside of it.
DSLs stand in contrast to general-purpose langugages (**GPL**), which apply across all domains. These are the typical programming languages like Java, Python, and Haskell.
A domain is a targeted subject area of a program. Examples of domains are:

- Web design, with HTML as a DSL
- Data Management, with SQL as a DSL
- Modelling of Systems, with UML as a DSL

A domain does not have to necessarily encompass something broad. It could only be defined to be only for one hospital, making the DSL specialized for the particular environment. Thus, when designing a DSL for exact cover we do not have to consider all the other NP-Complete problems.

There are also different ways to implement DSLs. Generally, there are two broad approaches to creating them.
DSLs implemented via an an independent interpreter or compiler are known as *External Domain Specific Languages.* Examples include LaTeX and AWK.
DSLs created with an host language as a library are known as *Embedded Domain Specific Languages.* They can be more limitied due to the syntax of the host language, however, are far easier to create. Examples of that are jQuery, and React.
As we want to use Haskell to create the exact cover DSL, we will be using an embedded domain-specific language.

Besides there being two different kinds of DSLs, there are also different degrees of embedding one.

Shallow Embedding Shallow Embedding immediateley translates to the target language. E.g. an Haskell expression $a + b$ would be translated to a String like "$a + b$" containing that target language expression.

Deep Embedding Deep Embedding would build a data structure that reflects an expression tree. E.g. $a + b$ translates to the Haskell data structure

Add (Var "a") (Var "b"). This allows transformations like optimizations before translating to the target language.
We will mostly use shallow embedding for our DSL.

2.3 Haskell

Haskell is a purely functional programming language developed in 1987, named after the logician Haskell Brooks Curry. It is based on the lambda calculus, which means a function produces always the same output given the same input. This makes the functions in Haskell **referentially transparent**[3]. Haskell is also **statically typed** with type inference, **immutable values** (as is in line with the functional programming paradigm), and **lazy evaluation**[4].

Why is Haskell well-suited for creating DSLs?

1. Haskell has an **Expressive and Concise Syntax**, which makes it well-suited for defining domain-specific abstractions. This allows creating DSLs closely resembling the problem domain
2. The **Type System** provides a high level of safety and expressiveness. This can be leveraged to create DSLs that enforce specific constraints.
3. **Monads** provide a powerful abstraction for dealing with effects and side effects. This can be used to create DSLs that capture complex computation patterns in a modular and composable manner.

2.3.1 Monads

Monads in functional programming provide a way to structure and manage computations with side effects while perserving referential transparency and composability. This is made possible by sequencing computations using the **bind** operation and wrapping values using **return**.

return: This operation lifts a value into the monad. It takes a pure value and wraps it inside the monadic context.

```
return :: Monad m => a -> m a
```

bind (or »=): This operation combines a monadic value with a function that takes a pure value and produces a monadic result. It allows sequencing of monadic computations.

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

This allows the programmer to control the order of execution and handle side effects.

[3]This stands contrary to an imperative language where the code can produce side effects that can influence the behavior of an application

[4]Lazy evaluation means that expressions are not evaluated until their results are needed

3 Related Work

3.1 The Art of Computer Programming, Volume 4, Fascicle 5

"The Art of Computer Programming" is a multi-volume work written by the author Donald Knuth. Donald Knuth is a computer scientist and mathematician who is widely known for his contributions to the field of computer programming, particularly in the area of algorithms and data structures, and his research is the corner-stone to this thesis.
Volume 4, Fascicle 5 of The Art of Computer Programming, released in November 2019, talks about the exact cover problem and introduces *Algorithm X*. Algorithm X uses recursion and backtracking to find all solutions of to some exact cover problem.

3.1.1 Dancing Links

Algorithm X main concept is **dancing links** (DLX) which is why the algorithm is also often referred to as DLX. Dancing links is a technique for adding and deleting a node from a circular doubly linked list.
A **doubly linked list** is a linked data structure that consists of a set of sequentially linked nodes. Each node contains three fields. One field for the value of the node, one field for linking to the previous node, and one field for linking to the next node. If the previous node of the first node is the last node, and the next node of the last node is the first node, then the doubly linked list is called circular.

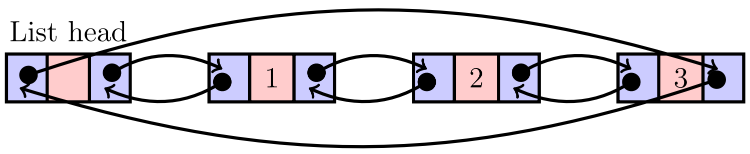


Figure 3.1: *Circular doubly linked list*

Dancing links uses the linked structure of the circular doubly linked list to its advantage.
The main ideas are these:

If x has to be deleted from the doubly linked list:

$$\begin{aligned} &\text{x.prev.next} \leftarrow \text{x.next} \\ &\text{x.next.prev} \leftarrow \text{x.prev} \end{aligned} \tag{3.1}$$

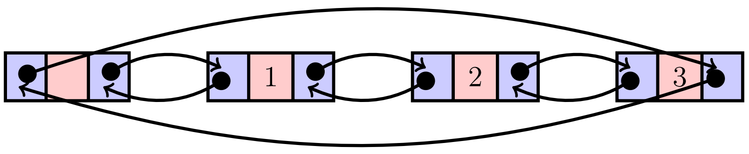
If x has to be restored (assuming that x.next and x.prev have been left unmodified):

$$\begin{aligned} &\text{x.prev.next} \leftarrow \text{x} \\ &\text{x.next.prev} \leftarrow \text{x} \end{aligned} \tag{3.2}$$

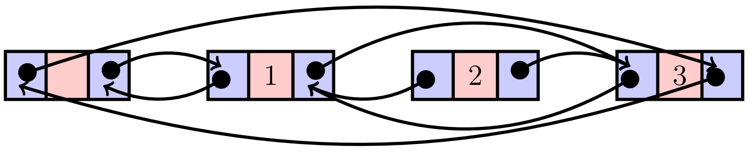
The adding and removing process of an item to every part of the doubly linked list is very fast (O(1)), especially if the item in question has to be restored after being removed. This makes this data structure very efficient for backtracking algorithms.

The choreography that underlines the motions of the pointers tangling and untangling themselves gives dancing links its name. Let us look at an example of it.

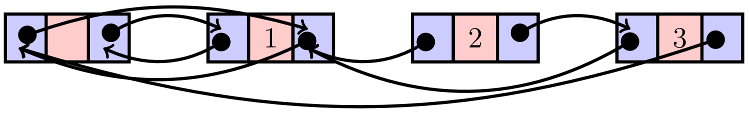
We start with an 3-element untangled list:



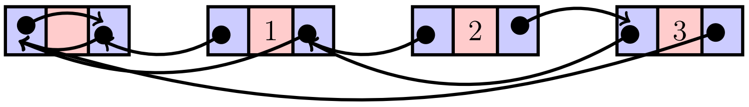
If we use (3.1) to delete element "2", the list becomes



And if we now decide to delete element "3", we get



And lastly, if we delete the last element "1", we have



The list is empty, and the links have become rather tangled. To untangle this list per hand would be quite the chore. However, because we know the order of how we deleted the elements ($2 \rightarrow 3 \rightarrow 1$), we can backtrack and restore the list to its initial state by restoring the elements with (3.2) in the opposite order.

3.1.2 Algorithm X

Algorithm X, or DLX, is used to solve the exact cover problem. To recount the main points from Chapter 2.1 Exact Cover:

- Problems are considered *exact cover* when each element of a set X has to be covered *exactly once.* These elements are called **primary items**
- Sometimes one needs an constraint that has to be covered *at most once.* These elements are called **secondary items**
- The subsets of X which represent our available choices, are called **options**, while the elements of X are called **items**

The basic idea is to create a recursive algorithm based on the operation of *covering an item.* Covering an item i means that we delete all of the options that contain i and we delete it from the list of items that need to be covered. Thus we have such a basic algorithm:

- Select an item i that needs to covered; terminate successfully if none are left (solution found)
- If no active options involve i, backtrack or terminate unsuccessfully (there is no solution). Otherwise cover item i.
- For each just deleted option O that involves i, one at a time cover each item $j \neq i$in O, and solve the residual problem.

When backtracking we need to of course uncover the reinstated item by recovering all options that its covering deleted and uncovering each item of these options.

Data Structure

Now the interesting part is what kind of data structure we are going to use for storing our items and options. Because we want to add and delete options and items efficiently we want to use dancing links. However, instead of using pointers or references we will be using an array which will serve as our doubly linked list. Each item of the array will have an index reference to their corresponding previous and next item.
We want to have two separate arrays. One for the items, and one for the options. The item array will have a list head at the start, so that if all items are covered, we would be able to check that by testing if the list head points to itself/a secondary item (we will come back to this later).

```
def item {
  string name;     - symbolic identification of the item
  int prev, next; - neighbors of this item
}
```

For the options array we have two different ways of implementing the nodes of the options.

1. Each node has a pointer to the node that is directly above and below them (the next option that contains this item), and to the previous node (left) and the next node (right) of the option.

   ```
   def node {
      int itm;
      int up,down;
      int left ,right;
   }
   ```

   This implementation uses a doubly linked list for the options, like we do for the items. This is an intuitive solution, but we can do better.

2. The left and right pointer of the nodes in the options never change. So we do not need the pointers.([**knuthwebsite**])

   ```
   def node {
      int itm;
      int up,down;
   }
   ```

   Instead we can go up or down in the array index to get to the left/right node. However, the question arises how we know when the option ends when we go traverse the option.
   This is solved by introducing *spacer nodes* at the end of options. They are recognized by their *itm* value, which is below 0. Their up field point to the start of the preceding option, while their down field point to the end of the following option. Thus it is easy to traverse an option circularly, in either direction.

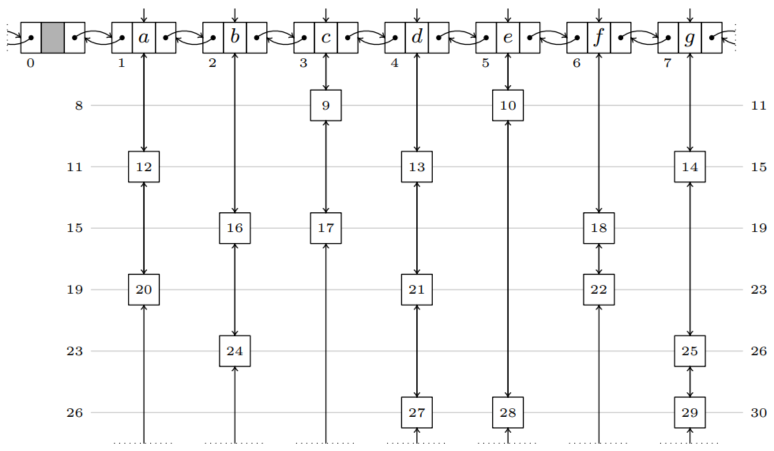


Figure 3.2: *Datastructure example for DLX [***knuthwebsite***]*

The numbers in 3.2 represent the indices of the option array. The first row indices of the option array are all the elements in the item array (of course the

values of the cells are differ in both arrays). In the option array the first row *itm* values do not refer to the corresponding itm index (because their index **is** the itm index). Instead it is the length of the remaining options which posses the item. This is used for deciding which item should be covered next (with the MRV heuristic).
The numbers to the left and right of the options are the spacer nodes. As an example the *down* and *up* values of spacer of index 11 in this case are respectively 9 and 14.

Algorithm

Let us first define the process of how the algorithm covers the options when covering an item (copied from [**knuthtaocp**]):

$$\text{cover(i)} = \begin{cases} \text{Set p} \leftarrow \text{down(i). (Here p, l, and r are local variables.)} \\ \text{While p} \neq \text{i, hide(p), then set p} \leftarrow \text{down(p) and repeat.} \\ \text{Set l} \leftarrow \text{left(i), r} \leftarrow \text{right(i),} \\ \quad \text{right(l)} \leftarrow \text{r, left(r)} \leftarrow \text{l.} \end{cases} \tag{3.3}$$

In 3.3 we see how for all options containing item i the procedure *hide* is called. After all the options are hidden the item is removed from the doubly linked list.

$$\text{hide(p)} = \begin{cases} \text{Set q} \leftarrow \text{p + 1, and repeat the following while q} \neq \text{p:} \\ \quad \text{Set x} \leftarrow \text{itm(q), u} \leftarrow \text{up(q), d} \leftarrow \text{down(q);} \\ \quad \text{if x} \leq \text{0, set q} \leftarrow \text{u (q was a spacer);} \\ \quad \text{otherwise set down(u)} \leftarrow \text{d, up(d)} \leftarrow \text{u,} \\ \qquad \text{len(x)} \leftarrow \text{len(x)} - \text{1, q} \leftarrow q+1 \end{cases} \tag{3.4}$$

hide(p) traverses the option containing p and removes all its elements from the doubly linked list that are the up and down fields. Each time an element is removed the length of that item is reduced by 1.

Uncovering an item works very similar to covering it. The only difference is that instead of removing an element we reinstate it. We also move in the opposite direction: In uncover we move not downward, but upward, and in unhide we move left instead of right.

$$\text{uncover(i)} = \begin{cases} \text{Set l} \leftarrow \text{left(i), r} \leftarrow \text{right(i),} \\ \quad \text{right(l)} \leftarrow \text{i, left(r)} \leftarrow \text{i.} \\ \text{Set p} \leftarrow \text{up(i).} \\ \text{While p} \neq \text{i, unhide(p), then set p} \leftarrow \text{up(p) and repeat.} \end{cases} \tag{3.5}$$

$$\text{unhide(p)} = \begin{cases} \text{Set q} \leftarrow \text{p} - 1\text{, and repeat the following while q} \neq \text{p:} \\ \quad \text{Set x} \leftarrow \text{itm(q), u} \leftarrow \text{up(q), d} \leftarrow \text{down(q);} \\ \quad \text{if x} \leq 0\text{, set q} \leftarrow \text{d (q was a spacer);} \\ \quad \text{otherwise set down(u)} \leftarrow \text{q, up(d)} \leftarrow \text{q,} \\ \qquad \text{len(x)} \leftarrow \text{len(x)} + 1\text{, q} \leftarrow q - 1 \end{cases} \tag{3.6}$$

Additionally to the cover and uncover procedures, we need for the algorithm a way to select the next item to be covered. For this we use the **Minimum Remaining Values** (**MRV**) heuristic:

$$\text{MRV} = \begin{cases} \text{Set } \theta \leftarrow \infty,\ p \leftarrow \text{right(0).} \\ \quad \text{While } p \neq 0\text{, do the following: Set } \lambda \leftarrow \text{len(p);} \\ \quad \text{if } \lambda < \theta \text{ set } \theta \leftarrow \lambda,\ i \leftarrow p\text{; and set } p \leftarrow \text{right(p).} \\ \quad \text{(We can exit the loop immediately if } \theta = 0\text{)} \end{cases} \tag{3.7}$$

Now that we have all the necessary functions, we define the *DLX* algorithm like this:

Algorithm 1 DLX

Input: Items and Options
Output: Solutions

```
X1. [Initialize.] Set the problem up in memory, as in 3.2. Also set $N$ to the number of items, $Z$ to the last spacer address, and l $\leftarrow$ 0.

X2. If right(0) = 0 (hence all items have been covered), visit the solution that is specified by $x_0x_1 \cdots x_{l-1}$ and go to X8.

X3. [Choose i.] At this point the item $i_1, \cdots, i_l$ still need to be covered, where $i_1 =$ right(0), $i_{j+1} =$ right($i_j$), right($i_t$) = 0. Choose one of them, and call it i. (Use the MRV heuristic described in 3.7)

X4. [Cover i.] Cover item $i$ using 3.3, and set $x_l \leftarrow$ down(i).

X5. [Try $x_l$.] If $x_l = i$, go to X7 (we have tried all options for $i$). Otherwise set $p \leftarrow x_l + 1$, and do the following while $p \neq x_l$: Set $j \leftarrow$ top(p); if $j \leq 0$, set $p \leftarrow$ up(p); otherwise cover(j) and set $p \leftarrow p + 1$. (This covers the items $\neq$ i in the option that contains $x_l$.) Set $l \leftarrow l + 1$ and return to X2.

X6. [Try again.] Set $p \leftarrow x_l - 1$, and do the following while $p \neq x_l$: Set $j \leftarrow$ top(p); if $j \leq 0$, set p $\leftarrow$ down(p); otherwise uncover(j) and set $p \leftarrow p - 1$. (This uncovers the items $\neq$ i in the option that contains $x_l$, using the reverse of the order in X5.) Set $i \leftarrow$ top($x_l$), $x_l \leftarrow$ down($x_l$), and return to X5.

X7. [Backtrack.] Uncover item $i$ using 3.5.

X8. [Leave level $l$] Terminate if $l = 0$. Otherwise set $l \leftarrow l - 1$ and go to X6.
```

This algorithm will be able to solve all exact cover problems that contain only primary constraints (, but theoretically also problems with secondary constraints, as these can be converted to primary ones). However, if we want it to also take secondary items into its input, we need to slightly change the algorithm.
We simply need to divide the items into two groups: The items that need to be covered *exactly once* (aka primary items), and the items that need to be covered *at most once* (aka secondary items). We then modify step X1 so that only the primary items appear in the active list which needs to be covered. We then just need to be careful the MRV heuristic ignores the secondary items, and everything should work without a problem.

3.2 Lazy Functional State Threads

The whole thing about *dancing links* is that the doubly linked list needs to be constantly updated. Yet Haskell creates only immutable values because of its no side effects policy. One way to create the DLX algorithm would be to constantly remake the doubly linked lists whenever a cell is updated. This is of course highly inefficient and the implementation would be only able to solve the simplest of problems in a timely manner.
A better way would be to use the **I/O-Monad** with the data structure *IOArray*. With this method we would be able to create a mutable array and implement it in an efficient way. The problem with this approach is that I/O is not referentially transparent, and the endproduct would not be **modular**. If we start using IO in this function, then everything else surrounding and using it would also become I/O.
Clearly this is also not optimal. However there is another way.

3.2.1 State Threads

The paper "Lazy Functional State Threads" by John Launchbury and Simon L Peyton Jones written in the Univerity of Glasgow on March 10, 1994. introduces **State Threads** which allow users to create stateful application without sacrificing a lot of computation costs, or relying on I/O. This is done by encapsulating the stateful computation in a *State Transformer*. This State Transformer computes the program and gives out a pure output, appearing as a referential transparent function to the rest of the program.

State

A system is described as **stateful** if it is designed to remember preceding events or user interactions. The remembered information is called the **state** of the system. The state of the system may change the behaviour of the function, such as this:

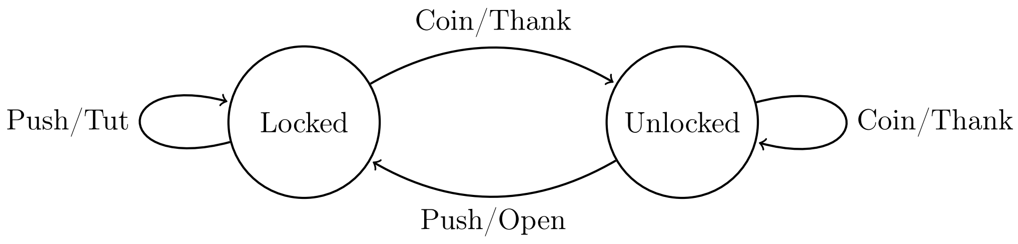


Figure 3.3: *Turnstile with States*

In 3.3 the two nodes are the states that a turnstile can have, and the labels of the edges have the input on the left side, and the response on the right side. Depending on which state one is in, the output can change (but not necessarily as one sees with the *Coin* input).

Stateful computations are also used to manipulate updatable state. So state can also be some mutable value, like an array. This is useful in our case as we need an array for the DLX algorithm. We will be able to change the array without needing to recreate it from scratch. For that we will use *State Transformers*

State Transformers

The paper introduces a way to express a stateful algorithm in a non-strict, purely-functional languages. It offers:

- **[Complete referential transparency]** How is this possible then state relies by definition on a I/O interaction to work? This works, because a stateful computation is a *state transformer*, that is, a function from an **initial state to a final state**. It is like a script detailing the actions to be performed on its input state. Because the script is a deterministic sequence of commands, the state transformer is a pure function.

- **[Encapsulating stateful computations]** It is possible to *encapsulate* stateful computations so that they appear to the rest of the program as pure (stateless) functions. Complete safety is maintained by this encapsulation. A program may contain an arbitrary number of stateful sub-computations, each simultaneously active, without concern that a mutable object from one might be mutated by another.

- **[Naming of mutable objects]** This gives the ability to manipulate multiple mutable objects simultaneously

- **[Lazy computations]** without losing safety. If we need only the first elements of some list that is computed by the state transformer, then only so many stateful computations are executed to produce these elements

- **[First class value]** It can be passed to a function, returned as a result, stored in a data structure, etc.

The state transformer has a value of type **ST s a**. This computation transforms a state indexed by type **s**, and delivers a value of type **a**. It is thought of like this:

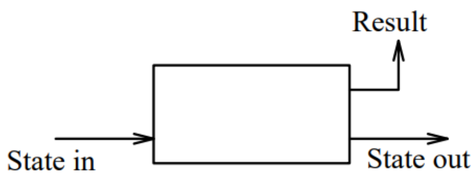


Figure 3.4: *State Transformer [***state***]*

This is a purely-functional account of state. The **ST** stands for "a state transformer", which is synonymous with "a stateful computation". Additionaly to getting an output to a given state, the ST also transforms the state into another.

A state transformer can also have other inputs besides the state, and have more than one output, like a function of this type:

$$\text{Int} \rightarrow \text{Int} \rightarrow \text{ST s (Int,Bool)}$$

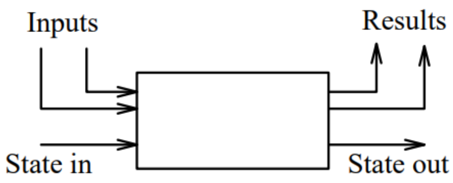


Figure 3.5: *State Transformer multiple Inputs and Outputs* [**state**]

The simplest state transformer, **returnST**, simply delivers a value without affecting the state:

$$\text{a} \rightarrow \text{ST s a}$$

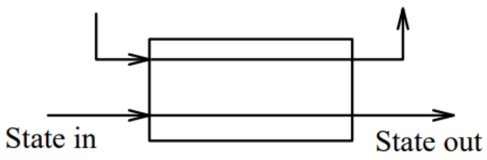


Figure 3.6: *returnST* [**state**]

Furthermore, we need the ability to compose state transformers in sequence. This can be done by using the function:

$$\text{thenST} :: \text{ST s a} \rightarrow (\text{a} \rightarrow \text{ST s b}) \rightarrow \text{ST s b}$$

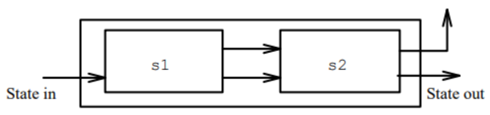


Figure 3.7: *thenST* [**state**]

Since ST is a monad, we can also simply use the *do notation* to compose several state transformers into one.

Finally we got:

$$\text{runST} :: \text{ST s a} \rightarrow \text{a}$$

This allows us to use state transformers a part of a larger program which does not manipulate state at all. The idea is that **runST** takes a state transformer as its argument, conjures up an initial empty state, and applies the state transformer to it. The result is returned while the final state is discarded.
To alleviate the issue of one thread being used in another runST is a **rank-2 polymorphic type**.

$$\text{runST} :: \forall \text{ a. } (\forall\text{s. ST s a}) \rightarrow \text{a}$$

This allows us to mitigate the issue where the reference allocated in one runST's thread is used inside another ones. If this type would not exist, then the outputs would not be deterministic, and could allow for different outputs depending on which thread goes first. This is of course something to be avoided.

3.2.2 STArray

The state transformers that we looked at before can only manipulate one item at once. For our purposes, we want the state transformer to manipulate an array, which can be thought of as multiple items, each independently mutable. Such an array can be created an manipulated with these functions:

$$\text{newArray} :: \text{(Ix i)} \implies \text{(i,i)} \rightarrow \text{a} \rightarrow \text{ST s (STArray s i a)}$$
$$\text{readArray} :: \text{(Ix i)} \implies \text{STArray s i a} \rightarrow \text{i} \rightarrow \text{ST s a}$$
$$\text{writeArray} :: \text{(Ix i)} \implies \text{STArray s i a} \rightarrow \text{i} \rightarrow \text{a} \rightarrow \text{ST s ()}$$
$$\text{modifyArray} :: \text{(Ix i)} \implies \text{STArray s i a} \rightarrow \text{i} \rightarrow \text{(a -> a) -> ST s ()}$$

While in many languages an array has to be indexed by integers, in Haskell you can use many different types as long as they can be mapped onto integers (this is what Ix i means). With these functions we can implement the *DLX* algorithms *doubly linked lists* efficiently.

4 Implementation DLX

This part of the thesis will be about the implementation of Donald Knuth's DLX algorithm [**knuthtaocp**].

4.1 DLX

In this section we will explain the implementation of the described *Algorithm X* in Chapter 3.1 to Haskell. Because Haskell is quite different to C, the language that Donald Knuth used, we needed to do some modifications to the original implementation that is uploaded on his website [**knuthwebsite**]. However, because of the **State Transformer-Monad**, or **ST-Monad** in short, we do not need to do major revamps of the code. The modification will be mostly bringing over global variables into the function arguments.

4.1.1 Doubly Linked Lists

First of all, we want to build the data structure for the backtracking part of DLX. We want linked items in one array, and options nodes with spacer nodes in between options in the other one. An example of that is 3.2.
The cells of the **item array** will contain the data structure

```
data Item a = Item {
  name :: a,
  prev :: Int,
  next :: Int
}
```

and the **option array** will have the data structure

```
data Node = Node {
  itm :: Int,
  up :: Int,
  down :: Int
}
```

As explained in Chapter 3.1 the items will be circularly linked with each other with one header item at the start. The header item would link to the first and last item with his pointers, but would not have a name. In C we can leave it uninitialized, however, in Haskell we cannot do that. Furthermore, because the name has a polymorphic type, we are not able to give it a base value (like "" for String or 0 for Int). Instead we let the first item share its name with the header item.
The procedure for the **item array** is shown in 2 The if-case could be also

Algorithm 2 Create Item STArray

```
    Input: items                                  ▷ primary and secondary Items
    Output: itemArr                               ▷ Item STArray
  Set $l$ to empty list
  l.append(Item {prev = 0, name = first item, next = last item})     ▷ header
  for $i = 1$ to length items do
      if last $i$ then
          l.append(Item {prev = i-1, next = 0, name = items[i-1]})
      else
          l.append(Item {prev = i-1, next = i+1, name = items[i-1]})
      end if
  end for
  Create STArray from l
```

summarized to "l.insert(*Item {prev* = $i-1$, *name* = *items*$[i-1]$, *next* = $(i+1)$ *mod (length items*$+1$*)*}".

The procedure for the **option array**, or **node array**, is quite a bit harder, since we have to connect all the nodes that share the same item, as well as link the spacer nodes to the first node of the previous option, and last node of the next option. The basic idea for the option nodes is to group all the nodes that have the same item, and link them together. For the spacer nodes we are going to create them in between options, group them up, and update *up* and *down* to the **next int** of the prev spacer node, and **prev int** of the next spacer node respectively. The algorithm is shown in 3

4.1.2 Algorithm

The DLX algorithm is described in Chapter 3.1 and can be implemented mostly as written. The only slight problem is that we cannot divide the algorithm in parts as easily in **Haskell** because we do not have access to global variables, and have a static type system. That is why the code has to be a bit more interconnected when for example in **C**.

Our DLX program takes *items*, and *options* as input. The *items* are divided into a tuple of lists of *primary items* and *secondary items*, while options are a list of list of elements. Each list represents one option. These inputs are used in **X1** to generate the **item array** 2 and the **node array** 3. Afterwards, the algorithm goes through steps **X1** to **X8** until it reaches **X8** with l = 0 (condition to terminate). The output is a list of all solutions. A solution consists of a list of options (the options are, in this case, only one index of one of their elements) that produces an exact cover to the problem.

As the options in the solution are only respectively one integer index from the node array, we implemented a quality of life function that finds all other elements of the the options, and converts the indices of the elements to their

Algorithm 3 Create Node STArray

```
Input: option, itemArr
Output: nodeArr                                        ▷ Node STArray

1. Set $l$ to empty list
2. Append all **items** in itemArr (converted to nodes) to $l$. Set $itm = 0$ for
   all of them (for item nodes $itm$ is used to represent the amount of nodes
   that share the same item as this node)
3. Append all **option elements** to $l$. Set $itm = i$, where $i$ is the index of
   the item in itemArr which matches the **option element**. Put **spacer
   nodes** in between every option, and at the start and end. Set $itm < 0$
   so that spacer nodes can be identified
4. **Group** all spacer nodes and **modify** $down = next\ spacer - 1$, $up =
   last\ spacer + 1$ (They do not have to be circularly connected)
5. **Group** all non-spacer nodes after itm (item nodes have to be added to
   the respective groups, since their $itm = len$), **connect** the grouped nodes
   circularly together, **update** $itm$ of item nodes
6. Create array from $l$
```

name.

Algorithm 4 Interpret DLX Solutions

```
Input: itemArr, nodeArr, solutions
Output: interpreted Solutions
Set $sols$ to empty list                                   ▷ solutions
for $is$ in $solutions$ do
    Set $sol$ to empty list                     ▷ interpreted solution
    for $i$ in $is$ do
        $j \leftarrow$ find start of option $i$ ▷ index of $left\ spacer\ node + 1$)
        $opt \leftarrow$ from $j$ to right spacer node add all elements to a list $l$,
                while converting their index to their respective names
        sol.append($opt$)
    end for
    sols.append($sol$)
end for
return $sols$
```

To convert an index i of an element to the name it represents, we only have to take the itm of the nodesArray[i]. This gives us the index j of the item and we are able to get the name from itemsArray[j].

Lastly, we explained in Chapter 3.1 how to incorporate secondary items into the algorithm, however have not shown how the pseudo code has to be modified. **X1** is modified from

X1. [Initialize.] Set the problem up in memory, as in 3.2. Also set N to the number of items, Z to the last spacer address, and l $\leftarrow$ 0.

to

X1. [Initialize.] Set the problem up in memory, as in 3.2. Also set N to the number of primary items, and l $\leftarrow$ 0. (We changed N from *items* to *primary items*, and deleted Z since we do not use it.)

We also need to change **MRV** 3.7 since it selects the next item to be covered.

```
Set $\theta \leftarrow \infty$, $p \leftarrow$ right(0).
    While $p \neq 0$, do the following: Set $\lambda \leftarrow$ len(p);
    if $\lambda < \theta$ set $\theta \leftarrow \lambda$, $i \leftarrow p$; and set $p \leftarrow$ right(p).
    (We can exit the loop immediately if $\theta = 0$)
```

$$\Longrightarrow$$

```
Set $\theta \leftarrow \infty$, $p \leftarrow$ right(0).
    While $p \neq 0$ or $p \geq n$, do the following:
    Set $\lambda \leftarrow$ len(p);
    if $\lambda < \theta$ set $\theta \leftarrow \lambda$, $i \leftarrow p$; and set $p \leftarrow$ right(p).
    (We can exit the loop immediately if $\theta = 0$)
```

MRV does not select an item if it is not a primary one, by filtering out the items that have an index higher or equals of n.

4.2 Finding solutions with DLX

We are going to explore here the normal way to solve a problem with the DLX algorithm.

4.2.1 Basic Examples

First we look at some basic examples to see how DLX works.
For example let us take the real world instance that is shown in 1.1. We have got the **primary items** *A, B, C, D, E, F*, which we are going to input as a list of strings to DLX. The **secondary item** list is going to be empty .
Next, we have the **options**:

A, B
A, B, C
C, E
D, F
E, F

Each option is going to be a separate list in a list of options with the elements being strings. Now we input the **items** and **options** into the **DLX algorithm** and get

A, B
C, E
D, F

as the **solution** in a list (all solutions) of exact cover options (list of lists of strings).

Another simple example, this time to illustrate how secondary items work, would be a problem with the items *A, B, C*, where *A, B* are the **primary items**, and *C* is a **secondary item**. The **options** are:

A, B
A, B, C

If we input this problem to the algorithm, we get two solutions:

Solution A	Solution B
A B	A B C

Table 4.1: *Solutions for 4 Queens problem*

Because secondary items are not needed, we have one solutions where the secondary item is present, and one where it is not.

4.2.2 Queens

We already talked about the Queens problem in Chapter 2.1. To summarize, we want to know all solutions to the problem of placing n queens on a nxn board without them attacking each other. As they can move horizontally and vertically, all the queens have to be on a separate row and column. Because there are n rows and columns, as well as n queens, each **row and column** has to be present **exactly once** (**primary items**). Each type of diagonal (upward/downward) is present more than n times on the board. However, all the queens have to be on separate diagonals, as they can move diagonally. This means that they the **diagonals** are **secondary items**.

We want to create an algorithm that gives us a **DLX-Format** for the **n Queens** problem. The constraints are described in 2.1 (row constraint), 2.2 (column constraint), 2.3 (upward diagonal constraint), and 2.4 (downward diagonal constraint). The procedure to **constructing the Format** looks like this:

Algorithm 5 Queen DLX

```
    Input: n
    Output: ((primary items, secondary items), options)

 1. Set i,j to 0, Set options to empty list
 2. Construct the primary items by iterating from 1···n while adding each
    number to the string "r" and "c"
 3. Construct the secondary items by iterating from 2···(2·n) while adding
    each number to the string "a" and "b"
 4. $i++$; If i > n Then output((primary items, secondary items), options)
    Else $j = 1$; go to 5.
 5. If j > n Then goto 4. Else options.append($r_i\ c_j\ a_{i+j}\ b_{n-i+j+1}$), repeat
    5.
```

We changed the downward constraint slightly and adjusted it to the upward constraint, so that both share the same numbers.

Now we can input every natural number for n, and this algorithm will give us the format needed for DLX. So if we want to get all solutions to the 8-Queens problem, we input *dlx (queens 8)* (the 8 is the n) and get all 92 solutions.
Sometimes, we want to have an easy way to understand how a solution looks. For this we can design a small algorithm that helps us **interpret the solutions** given to us by the DLX algorithm.

Algorithm 6 Interpret Queens-Solutions

```
  Input: n, solutions                    ▷ Int, [[[String]]]
  Output: solutions                      ▷ String
Set res to ""
for sol in solutions do
    for row in sol do
        for $j = 0$ to $n$ do
            if $i = j$ then
                res.append("Q ")
            else
                res.append("- ")
            end if
        end for
        res.append(linebreak)
    end for
    res.append(linebreak)
end for
```

With this algorithm we can make the solutions more graphic. For example if we do not use the algorithm, our solutions would look like this:

Solution A	Solution B
r_1 c_2 a_3 b_6	r_1 c_3 a_4 b_7
r_2 c_4 a_6 b_7	r_2 c_1 a_3 b_4
r_3 c_1 a_4 b_3	r_3 c_4 a_7 b_6
r_4 c_3 a_7 b_4	r_4 c_2 a_6 b_3

Table 4.2: *Format solutions for 4 Queens problem*

If we use **interpretQueens** on the problem it would look like this:

Solution A	Solution B
- Q - -	- - Q -
- - - Q	Q - - -
Q - - -	- - - Q
- - Q -	- Q - -

Table 4.3: *Graphic solution for 4 queens problem*

Later on in this thesis we are going to generalize this algorithm, so it can convert the solutions to many problems into a more digestible output.

4.2.3 Sudoku

Sudoku is a logic-based, combinatorial number-placement puzzle, that is accidentally also an exact cover problem. In classic Sudoku, the objective is to fill a 9 x 9 grid with digits so that each column, each row, and each of the nine 3 x 3 subgrids that compose the grid (also called **boxes**) contains all of the digits 1 to 9.

	2		5		1		9	
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Unsolved Sudoku

4	2	*6*	5	*7*	1	*3*	9	*8*
8	*5*	*7*	2	*9*	3	*1*	*4*	6
1	3	*9*	*4*	6	*8*	*2*	7	*5*
9	*7*	1	*3*	*8*	*5*	6	*2*	*4*
5	4	*3*	*7*	*2*	*6*	*8*	1	9
6	*8*	2	*1*	*4*	*9*	7	*5*	*3*
7	9	*4*	*6*	3	*2*	*5*	8	*1*
2	*6*	*5*	8	*1*	4	*9*	*3*	7
3	1	*8*	9	*5*	7	*4*	6	*2*

Solved Sudoku

Figure 4.1: *Example of a Sudoku púzzle*

We want to create a program that is able to create a **DLX format** based on an unsolved 9 x 9 Sudoku grid. For that we should first understand the constraints.

$$\text{Row constraint: } \sum_{i=1}^{n} \sum_{inp=1}^{9} \begin{cases} 1 & x_{ij} = inp \\ 0 & x_{ij} \neq inp \end{cases} = 1 \text{ for } 1 \leq j \leq n \tag{4.1}$$

$$\text{Column constraint: } \sum_{j=1}^{n} \sum_{inp=1}^{9} \begin{cases} 1 & x_{ij} = inp \\ 0 & x_{ij} \neq inp \end{cases} = 1 \text{ for } 1 \leq i \leq n \tag{4.2}$$

$$\text{Box constraint: } \sum_{j=1}^{n} \sum_{inp=1}^{9} \begin{cases} 1 & x_{box(i,j)} = inp \\ 0 & x_{box(i,j)} \neq inp \end{cases} = 1 \text{ for } 1 \leq i \leq n \tag{4.3}$$

$$box(i,j) = \left\lfloor \frac{i-1}{3} \right\rfloor \cdot 3 + \left\lfloor \frac{j-1}{3} \right\rfloor \tag{4.4}$$

The box function labels each 3x3 grid with one integer. The top left box is labeled as 0, the one to the right 1, the one below 0 is 3, etc. until we got 8 at the bottom right.

The idea for the algorithm is that we will have different items for positions, rows, columns, and boxes, and they all have to be present in each option. They will have this format/meaning:

- position: $p_{row,\ column}$
- row: $r_{row,\ inputValue}$

- column: $c_{column,\ inputValue}$
- box: $b_{box,\ inputValue}$

Row, column, and box items need to match on the *inputValue* in each option. We will create options only for empty cells. After figuring out what values are missing for the row, column, and box that the empty cell is in, the algorithm makes each combination where the *inputValues* match. Since there might be more than one option for each cell, we need the position item, so that **exactly one** option for each cell is selected.

Algorithm 7 Create Sudoku DLX format

```
Input: board                                    ▷ list of row of inputs
Output: ((primary items, []),options)

1. Define pos[9][9], row[9][9], col[9][9], box[9][9]
2. Check for errors in board (rows/columns/boxes have multiple of same
   input).
3. Save the filled in cell positions as pos[i][j] = d+1, row[i][d] = j+1,
   col[j][d] = i+1, box[box(i,j)][d] = i+1 (see 4.4), where
   $i \in row, j \in col, d \in inputs$ ($row/col$ = [0..8], $inputs$ = [1..9])
4. Create items: $\sum_{i=0}^{row}\sum_{j=0}^{col}$ For all not filled in values in pos[i][j],
   row[i][j], col[i][j], box[i][j] output $p_{ij}, r_{i(j+1)}, c_{i(j+1)}, b_{i(j+1)}$ as primary
   items
5. Create options: $\sum_{i=0}^{row}\sum_{j=0}^{col}\sum_{d=1}^{inputs}$ If pos[i][j] and row[i][d-1] and
   col[j][d-1] and box[box(i,j)][d-1] are not empty Then save option as
   $[p_{ij}, r_{id}, c_{jd}, b_{box(i,j)d}]$
```

We can input a board, like 4.1, into this algorithm as a list of strings, and program it to recognize an empty cell if the character is not "1"-"9".

Like with the n Queens problem, we want the solution to be more readable. We can create a small algorithm that takes the initial board, and inserts the options into all the unoccupied cells. To do this we go through each option, and use the coordinates given by p_{ij} to insert the input that the option carries. This input can be for example found in r_{id} as the d-value. We do this for each solution, and the solutions are easily understood.

Thus, instead of [[["p00","r04","c04","b04"], ["p02","r06","c26","b06"], ["p04","r07","c47","b17"], ["p06","r03","c63","b23"], $\cdots$]] we get something akin to the solved portion of 4.1.

5 DSL

The previous method of solving an exact cover problem is to create an algorithm that produces a format that you can input into the DLX algorithm to solve the problem. For people not knowing the programming language or even having no programming experience this is quite a hindrance. That is why we want to create a DSL that makes solving exact cover problems easy and intuitive.

5.1 Data Structures

The basic idea for the the DSL is to create a data structure where the user can input his problem and let it be solved. However, there are a lot of different types of exact cover problems that need different kinds of constraints, inputs and boards (if a board even exists). For example, we have **Polyominoes**. A polyomino is a rookwise-connected region of 5 square cells, and the goal is to put together the 12 possible shapes of a polyomino into a rectangular box. The creation of the format, as well as the input differs greatly from, for example, **Sudoku**. That is why we first want to generalize problems like **Queens**, and **Sudoku** before tackling the bigger problem.

5.1.1 Inputs

We want our data structure to take in the possible **Inputs** for the problem. For Queens it might be Q, while for the standard Sudoku it would be 9 separate inputs *1-9*. There are problems where an input has to occur multiple times per shape, like the **Ian** problem, where the input *1* occurs once per row and column, *2* occurs twice, and so on until *4*. That is why we also need the ability to specify it.

```
data Input = InputsAndOccurences [(Int,String)]
           | ManuallySelectOccurence [String]
```

Input has two variants

- **InputsAndOccurences** takes a list of tuples as its argument. The tuples consist of the number the input occurs in a shape, and the inputs name.
- **ManuallySelectOccurence** takes a list of Strings as its argument. We input this if we want to manually input how often an input occurs in each constraint (notice that **InputsAndOccurences** defines it for all constraints)

To make the inserting of the inputs and their occurrences more intuitive, we have these 3 functions:

$$\textbf{occurs} \text{ input i} = [(\text{i, input})] \tag{5.1}$$
$$\textbf{occurEach} \text{ inputs i} = \text{map (i,) inputs} \tag{5.2}$$
$$\textbf{next} \text{ a b} = \text{a} ++ \text{b} \tag{5.3}$$

With this we can write for

- **Queens**: "Q" 'occurs' 1
- **Sudoku**: ["1",$\cdots$,"9"] 'occurEach' 1
- **Ian**: ("1" 'occurs' 1) 'next' ("2" 'occurs' 2) 'next' ("3" 'occurs' 3) 'next' ("4" 'occurs' 4)

5.1.2 Board

After this we have the board of the problem. Even though there are exact cover problems that are three dimensional or higher, for simplicity's sake we restricted the board size to two dimensions.

$$\textbf{type Board} = [[\text{String}]]$$

The user inputs here the initial state of the board. If the initial board is an empty rectangle, like for example in the Queens problem, we can use the function

$$\textbf{rectangle} \text{ x y} = [[\text{""} \mid \text{x} \leftarrow [1..\text{x}]] \mid \text{y} \leftarrow [1..\text{y}]] \tag{5.4}$$

An interesting thing that occurs is that we can restrict now a problems outputs by fixing an input in the initial board state. Let us take the 4 Queens problem and restrict the first input:

Board

[[""],["Q"],[""],[""],
[""],[""],[""],[""],
[""],[""],[""],[""],
[""],[""],[""],[""]]

$\Longrightarrow$

Solution

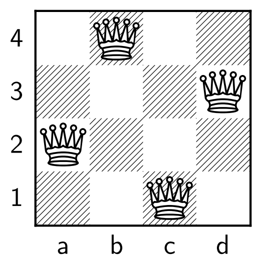


Table 5.1: *Restricted Solutions for 4 Queens Problem*

5.1.3 Constraints

Finally, we have **Constraints**. Every (exact cover) problem is defined by its constraints. **Queens** has (2.1), $\cdots$, (2.4), **Sudoku** (4.1), $\cdots$, (4.3). We want the user to be able to input the constraints that his **Inputs** and **Board** have to satisfy. It is defined as

data Constraints = Constraints [Int] | ManualSelect

where the Integers represent predefined constraints. For example, the **row constraint** that restricts how many of each input is allowed to be in each row, is defined as 1. So that the user does not have to constantly look up what number represents what constraint, we gave each constraint a name:

rowCons = [1]
colCons = [2]
downDiagCons = [3]
upDiagCons = [4]
boxCons = [5]
customCons = [6]

As previously said, 1, or **rowCons**, represents the row constraint, **colCons** the column constraint, **downDiagCons** the downward diagonal constraint, **upDiagCons** the upward diagonal constraint, **boxCons** the box constraint, and finally **customCons** a constraint where the user himself can give a shape which will be restricted.

5.1.4 Exact Cover

Finally putting it all together we got

```
data Exact_Cover = ECP {
        input :: Input,
        board :: Board,
        primaryCons :: Constraints,
        secondaryCons :: Constraints
}
```

primaryCons are the constraints that have to be covered exactly once.
secondaryCons are the constraints that have to be covered at most once.

If we would want to solve the **8 Queens problem**, we would define an instance like this:

$$\begin{aligned}\text{ECP } \{ &\textbf{input} = \text{InputsAndOccurences ("Q" 'occurs' 1)},\\ &\textbf{board} = \text{rectangle 8 8},\\ &\textbf{primaryCons} = \text{Constraints (rowCons 'next' colCons)},\\ &\textbf{secondaryCons} = \text{Constraints (downDiagCons 'next'}\\ &\qquad\qquad\qquad\qquad\text{upDiagCons)}\\ &\}\end{aligned} \tag{5.5}$$

5.2 DLX Formating

Now how do we get from a defined instance to a solution to the problem? Because we want to use DLX as our solver, we first need to create a **DLX Format** from **Exact_Cover**. The idea is to go from condition to condition creating for each one its own format, and then combining the same-cell ones with each other (only the items with same input are put together into the same option). This approach is sadly quite limited since it works only on problems where the options represent exactly one cell. Or more specifically, it works only on constraints that are based on some shape (row, column, box, etc.), and not, for example, on constraints that describe how inputs interact with each other (like a constraint that says that x inputs have to be adjacent). The algorithm can however be expanded to contain also different types of constraints.

5.2.1 Procedures

We first want to define the function to manually select how often an input appears in a constraint.

Algorithm 8 Manually Select Input Appearence

```
Input: inputs                                              ▷ [String]
Output: inputs                                             ▷ IO [(Int,String)]

1. For each Input, put in console how often it appears per shape
2. Ask for confirmation. If confirmed Then return Inputs with their oc-
   curence Else goto 1
```

Algorithm 8 is only used if the data structure **Input** is selected as **ManuallySelectOccurence** (see Section 5.1.1). This is used on every constraint.

Next we have a function that creates an item for each cell in the board and puts them in a separate list in a list of a list of String. The reason we need the nested lists is that when we create the format, we want to have all options for one cell in a list of lists, where each list represents one option. The following function is the base case for our format.

Algorithm 9 Create Board Positions

```
Input: transform, board    ▷ ([[String]] → [[String]]), [[String]]
Output: format             ▷ [[[String]]]
Set pos to empty list
for $i = 0$ to length board do
    for $j = 0$ to length board[i] do
        pos.append($p_{ij}$)
    end for
end for
Use transform on pos
Nest another list around each $p_{ij}$
```

The argument *transform* in Algorithm 9 is a function that moves around p_{ij}'s in such a way that matches the given constraint. **We want each row to contain one shape of the constraint**. For example, for the column constraint we would use the function *transpose*. That would move each column to a row. This makes it easy to create a format as will be shown next.

Algorithm 10 Create Options

Input: transformedBoard, inputs, baseFormat, letter
Output: format ▷ [[[String]]]

1. **Set** *options* to empty list
2. **For** *(occurs,input)* in *inputs*; do 3.-7.
3. **Save** how often input has yet to occur in each row of transformedBoard in *inputAmount*
4. **For** each element in *inputAmount*; Add all previous elements in *inputAmount* together (add a 0 to inputAmount beforehand). (scanl (+) 0 inputAmount)
5. **For** $i = 0$ to *length of transformedBoard*; **For** *elem* in *transfromed-Board[i]*; **If** *elem* is empty; **If** *itemDigits[i] - (ItemDigits[i+1]-1)* < 0 goto 6. **Else** goto 7.
6. **Designate** field as invalid (because if no options of this constraint can be constructed here, then no options of other constraints will be able to combine into a format on this cell (they need the same input for that (see Algorithm 11))).
7. **For** *j* in [itemDigits[i]..(itemDigits[i-1]-1)]; Append $letter_{j'-'input}$ to options (the position where it is saved has to correspond to the *elem* position in *transformedBoard*)
8. **Combine** *baseFormat* with *options* (with Algorithm 11)

The argument *transformendBoard* in Algorithm 10 is the board that was transformed with the same function that would be used in Algorithm 9.

Algorithm 11 Combine Format with Options

```
    Input: format, options                        ▷ [[[String]]],[[String]]
    Output: format                                ▷ [[[String]]]

 1. For the corresponding elements of format (f) and options (o); goto 2.

 2. If length $f = 1$ (only $p_{ij}$ item in format) or the input of $o$ matches with
    f[1] (to get input split on '-' and take the latter elem), Then f.append([o])
```

With Algorithm 9,10, and 11 we can construct formats for the constraints. Now we only need to combine the functions in one function.

Algorithm 12 Get DLX Format

```
    Input: constraints, board, inputs, prim
    Output: ((primary items,secondary items),options)
format ← Algorithm 9 with id, board as arguments
for con in constraints do
    transform ← Select transform function based on con      ▷ row:=id, column:=transpose, etc.
    if inputs are selected as ManuallySelectOccurence then
        inputs ← Algorithm 8
    end if
    subformat ←Algorithm 9 with transform, board as arguments
    subformat ← Algorithm 10 with (transform board), inputs, subformat, lt as arguments      ▷ lt starts with 'a', goes to 'b', ···
    Reposition the options in subformat to their original position $p_{ij}$
    Concat subformat and remove the position elements
    format ← Algorithm 11 with format, subformat as arguments
end for
options ← Concat format
items ← All unique elems of options; primary items ← Elems that were generated by first prim constraints; secondary items ← Rest
return ((primary items, secondary items),options)
```

With **Algorithm 12** we are now able to create a format for all problems with constraints where **each input i is allowed exactly/at most their *occurence rate* in a given shape**

5.2.2 Transform functions

As previously mentioned we have different transform function for different constraints, so that each shape of the constraint goes into a separate list/row. This is really the only difference then creating the format for the constraints that were mentioned in **5.1.3 Constraints**. We are going to quickly go through the different transform function:

- **[rowCons]** is the most basic one, because we do not have to make any changes to the baseFormat/board to get the right one
- **[colCons]** we use transpose
- **[downDiagCons]** we use a function *diagonals* that saves all the downward diagonals into a separate list
- **[upDiagCons]** we *reverse* the row positions and then use the funciont *diagonals*
- **[boxCons]** we use two nested for loops where the first for loop goes row wise and the second one column wise. Each iteration of the row loop should take the next h rows (h is defined as the height of the box), while each iteration of the column loop should take the next w columns (w is defined as the width of the box). Each box is $h \cdot w$ big, and they are enumerated from left to right, top to bottom.

The last one is the **customCons**. It is a bit different from the others, because it can do every shape there is (we can actually replicate any other constraint that was mentioned before with it). The algorithm looks like this:

1. **Set** *shapeIndices*, and *shapes* to empty list
2. **Select** a legal x-index and y-index for board; Append (x,y) to *shapeIndices*; Make (x,y) an illegal field
3. **If** done **Then** append *shapeIndices* to *shapes*; Set shapeIndices to empty list; goto 4. **Else** goto 2.
4. **If** last shape **Then** goto 5. **Else** goto 2.
5. **Create** a function *transform* where each list from *shapes* transforms the input (for example board) in a way that the shapes, described in each list of *shapes*, are put into separate lists; return *transform*

With this we can get a format (and a solution) for, for example, the **Jigsaw Sudoku** puzzle (5.1). The same rules apply for this puzzle as for a standard Sudoku puzzle, however, there are no boxes. Instead we have these bold-lined jigsaw regions. Each of those need to have all 9 numbers.
We just need to manually insert all the jigsaw regions with the **customCons**, use the **row-** and **colCons** for the rows and columns, and we would get a format with which we are able to solve the problem.

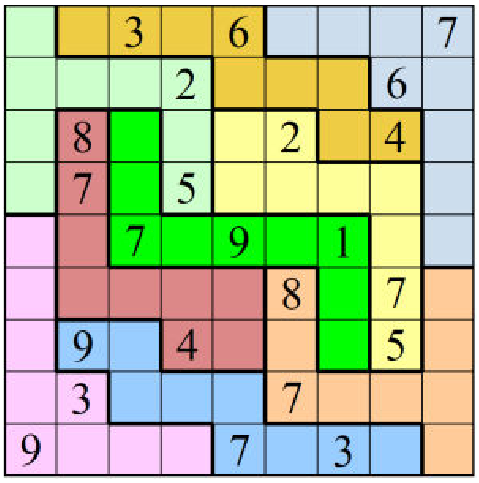


Figure 5.1: *Jigsaw Sudoku*

5.2.3 Quality of life

When we use the created format in the DLX Algorithm and get a solution, it would be nice to have a visual representation of what the solution looks like (like in 4.3). This is not so difficult to implement, as our format pattern is consistent when using our DSL. An option might look something like this: "p_{ij} a_{k-d} b_{k-d} $\cdots$". The i and j are the position indices of the board cell that the option represents. The a and b represent different constraints, k represents the item k of the constraint, and d is the input.
To get a visual depiction of a solution, we just have to take the initial board and insert all the options that we got from DLX into the respective p_{ij} cell with the input d.

Algorithm 13 Interpret Solution

```
    Input: board, solution                        ▷ [[String]],[[String]]
    Output: ()                                    ▷ IO ()
  Set result to empty list
  for option in solution do
      Insert in cell $p_{ij}$ of board Input $d$
  end for
  Print out board row by row, elem by elem
```

We can loop over all solutions while using **Algorithm 13**, and thus get all the solution in an easy to understand way.

We can also expand our **Exact_Cover** data structure definition with *showInitial :: Bool*, to show the initial board state before showing the solutions. We could also expand it with *showXSolutions :: Int* (shows only the first x solutions), *interpret :: Bool* (asks if we want to use **Algorithm 13**), and more.

5.3 Extensions for DSL

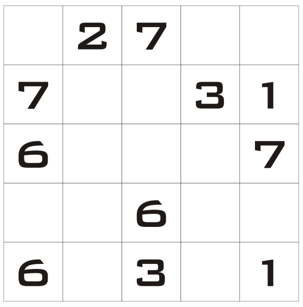
We mentioned previously that our DSL is pretty limited in what problems it is able to solve. We are able to extend the constraints of our DSL, however we are not able to use **Algorithms 10, 11,** and **12**. We need to create new methods to creating the options (of course it would be best if the methods would be as generally applicable as possible). We are going to show a problem that is not solvable with our DSL and then extend the DSL to make it solvable.

5.3.1 Fillomino

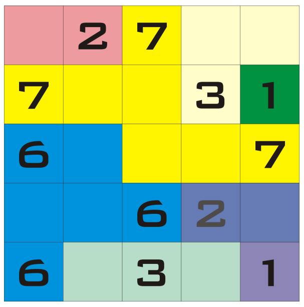
Fillomino is a logic puzzle which is played on a rectangular grid with no standard size. Some cells of the grid start with contained numbers, called **givens**. The goal is to divide the grid into regions called **polyominoes** such that each given number n in the grid satisfies the following constraints

- Each clue n is a part of a polyomino of size n
- no two polyominoes of matching size (number of cells) are orthogonally adjacent

It is possible for two givens with matching numbers to belong to the same polyomino in the solution, and for a polyomino to have no given at all.



(a) *Initial board*



(b) *Solution*

Figure 5.2: *Fillomino example*

We are not able to solve this problem because the constraints of this puzzle are not based on a shape but on the proximity of the inputs to one another. We need to create a separate method that can solve such constraints.

5.3.2 Connected Inputs Constraint

We add to our constraints

$$\textbf{connectedInputs} = [7]$$

The method we will use is described in detail in [**knuthwebsite**], but a rough summary is this:

1. For *d* in **Inputs** of the **Exact_Cover** data structure, do 2-3
2. **For** every cell in board **Do**: **If** cell is empty and adjacent to *input* Or cell is a *given* with wrong input **Then** goto next cell **Else** goto 3.
3. **If** cell is a *given* **Then** we backtrack if the *d*-omino gets too big, or if we are forced to choose a *d*-cell whose options have already been considered. **Else** we backtrack if we are next to a *d*-cell, or if solutions for this cell have already been considered

With this algorithm we are able to generate options that have *d* position-itemss (as primary items), that indicate which cells are looked at in this option, as well as some secondary items that check that no other *d*-omino is orthogonally adjacent to this one.
As such we extended our DSL to be able to solve problems like **Fillomino**.

6 Conclusion

In this thesis, we researched the topic of solving exact cover problems efficiently and intuitively. For this we first defined the types of constraints and exact cover problem might have. Namely *primary constraints*, which need to be covered *exactly once*, and *secondary constraints*, which need to be covered *at most once*. The state of the art **Algorithm X** by Donald Knuth is able to solve these exact cover problems by covering these constraints (items), checking if all primary constraints there covered, and backtracking if some constraints have been covered too many times. The backtracking is done via **dancing links**. This technique lets us delete and reinstate elements in constant time, while preserving the structure of the data.

Our main contribution lies in the creation of the domain-specific language in Haskell. It is able to solve a subset of exact cover problems while only needing a board, the inputs, and the constraints to be defined. This makes the creation of exact cover problems to be solved less restrictive and intuitive. Currently the DSL is only able to solve problems with constraints that are based on a shape of the board. This is done by reducing these constraints into a row-shaped constraint by transforming the board with an appropriate function. A problem was also shown that would not be solvable by the described DSL. However, a simplistic way to extend the language to be able to also capture the necessary constraints was presented, so that the problem might be solved.

This thesis serves as an introductory step towards creating a powerful domain-specific language that would help people with little to no programming knowledge create and solve all kinds of exact cover problems by experimenting with different boards, inputs, and constraints. However, as the DSL currently stands, it is not that powerful. As such, further research can be made to extend it by generalizing more constraints, or, if possible, generalizing all constraints. Another avenue of research can be the research of the extension of the **DLX Algorithm** with color-controlled covers, and how it might impact the DSL. Color control allows for more flexibility for non primary items, thus being another tool that might be helpful for generalizing constraints.

List of Figures