Rheinland-Pfälzische Technische Universität Kaiserslautern Landau

A Domain-specific Language for Recreational Mathematics

Bachelor's Thesis

by

Aurelijus Kadzys

December 29, 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 29.12.2023

Aurelijus Kadzys

Abstract

This thesis introduces a way of solving recreational problems in an easy-to-understand and intuitive manner. The particular problems that are looked at are called exact cover problems. They are problems where the constraints need to be (mostly) covered exactly once. Since this topic has already been covered in depth by others, there are already a lot of different solvers for such problems. One in particular, Algorithm X by Donald Knuth, is looked at in depth and worked around. However, for this algorithm to work, the constraints and options of the problem to be solved need to be identified and put into it.

Since writing out the constraints and options takes a lot of time, a procedure is often coded that does it for one. However, writing a procedure is not only a big time investment but also requires coding experience. We want to create a domain-specific language that is able to create a format for a problem without needing to code anything. It could be used by anybody who understands the problems constraints, and how the covering works. Not only would it make it easy to solve the specific kinds of problems, but it would also be a good tool to create new problems due to its straightforward application and availability of constraints, making it easy to experiment with different rules 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 because they challenge our logical thinking, pattern recognition, and, in some cases, memory, making it satisfying when we are finally able to solve a challenging puzzle. Sometimes, however, there is a need for an algorithm to either check if a solution is correct, find a solution, or test out ideas for a new puzzle.

The interesting part is that the aforementioned puzzles have a lot in common. Namely, they are all 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 challenge 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 valid ways to cover some cells. Since every cell can only be defined as one digit, we would need to discard all possible coverings besides one (**A.**). In this case, we would have multiple correct solutions. Another rule states that all cells in Sudoku need to be covered, or else it is not a valid solution (**B.**).

Exact cover problems are not limited 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 has competencies in architecture, building physics and chemistry; Charlotte has competencies in chemistry and electrical engineering; Dennis has competencies in data processing and financing; and Emma has competencies in electrical engineering and financing.

If we take Boris, then Charlotte would have to drop out since they overlap on C. To cover the remaining competencies D, E, and F, we would be 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. Therefore, we need to pick both Anna and Charlotte. Taking Charlotte makes Emma drop out, leaving only Dennis as

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*

the only other option. Because Dennis takes care of the last two competencies, we have a solution to the problem, as all the competencies are covered exactly once. To put it mathematically:

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

This specific problem might be quite easy to solve, however, what if there were two or three times the number of items/constraints and options? This would take a far longer time to solve and be quite complicated if we tried to use a more elaborate method than brute force.
This leads to a problem. Generally, problems that are difficult for humans are also likely to be difficult for computers. This applies in this scenario. Exact cover is known as one of Karp's 21 NP-complete problems[1]. A problem is classified as NP when it has a running time of **nondetermistic polynomial time**. This means that if we have a nondeterministic computation, the problem would be solvable in polynomial time (but for deterministic computations, the time might be exponential). Since, we do not have an answer to NP $\overset{?}{=}$ P we are not able to write polynomial-time algorithms for NP problems. Thus, we are only able to create an exponential time algorithm for exact cover. As such, we need an efficient algorithm, otherwise, the algorithm would only be able to solve the simplest of problems. **Donald Knuth** has designed such an algorithm for exact cover. His work is currently the state of the art for this problem.

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. The former is really tedious and prone to human error, while the latter requires the person to have coding experience. The goal for this thesis is to create a **domain-specific language** (**DSL**) that makes the process of creating a format for any given problem fast and easy while having no requirement to create an algorithm. The host language for the DSL is going to be Haskell, since it has ample tools for creating one.

[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

The term **exact cover** comes from the need to have each element in the set be covered exactly once. To be more specific, an 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 of $\mathcal{S}$.

Exact cover is as such, 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 exist that do not need to be covered but can also be covered 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 the algorithm to them because it saves memory and running time.

Let us take a look at the exact cover problem: **n Queens**. The goal 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, and the board rows and columns match the amount of queens, each of the queens has to be on different rows and columns/each row and column has to have exactly one queen on it. Hence, we need items that represent each row and column, and each of them has to be covered exactly once. These would be our **primary constraints**.
Queens are also able to 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, since that would mean that some queens are attacking each other. This makes the diagonal items **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 give the items specific names. 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). As an example: The option 'r_i c_j a_{i+j} b_{i-j}' represents a queen placed on cell X_{ij} which is on the upward diagonal a_{i+j} and downward diagonal b_{i-j}.
With n = 4, the placement options would be:

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:

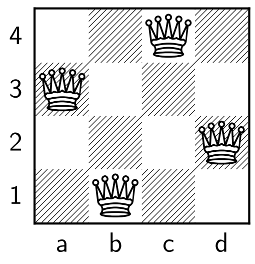
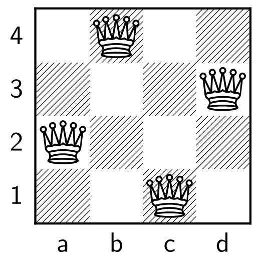
Solution A	Solution B
	
r_1 c_2 a_3 b_{-1}	r_1 c_3 a_4 b_{-2}
r_2 c_4 a_6 b_{-2}	r_2 c_1 a_3 b_1
r_3 c_1 a_4 b_2	r_3 c_4 a_7 b_{-1}
r_4 c_3 a_7 b_1	r_4 c_2 a_6 b_2

Table 2.2: *Solutions for 4 Queens problem*

Such problems (as with almost any problem) get 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 for a particular application domain. They are created to make solving problems in their field less difficult but are not intended to solve problems outside of it (but sometimes they still become Turing complete, like the combination of HTML and CSS).
DSLs stand in contrast to general-purpose languages (**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
- Modeling of Systems, with UML as a DSL

A domain does not have to necessarily encompass something broad. It could only be defined to be for one hospital, making the DSL specialized for the particular environment.

Generally, there are two broad approaches to creating these languages:
DSLs implemented via 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 limited due to the syntax of the host language; however, they are far easier to create since there is no need to create a whole interpreter or compiler for the system. Examples of that are jQuery and React.
Because we want to use Haskell to create the exact cover DSL, we will be using an embedded domain-specific language.

There are two different ways of creating an embedded domain-specific language.

Shallow Embedding immediately translates to the target language. E.g., a Haskell expression $a + b$ would be translated to a string like "$a + b$" containing that target language expression.

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.

For our purposes, we will be using a combination of shallow embedding and deep embedding. Deep embedding will be used to give the user more options when inputting their constraints and inputs, while shallow embedding will be used for the board and inputs.

2.3 Haskell

Haskell is a purely functional programming language developed in 1987 and named after the logician Haskell Brooks Curry. It is based on the lambda calculus, which makes functions always produce the same output given the same input. This leads Haskell to be **referentially transparent**[3]. Besides that, Haskell is **statically typed**, has **type inference**, possesses **immutable values** (as is in line with the functional programming paradigm), and is **lazy**[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 for the creation of 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.

Of course, there are also some downsides:

1. While Haskell is quite performant compared to languages like Python, It might not be the best choice for some performance-critical applications
2. Haskell has a bit of a niche community. As such, there is not as much support and help available as in mainstream languages

Nevertheless, Haskell is a powerful language with structured and sophisticated approaches to solving problems and creating DSLs.

2.3.1 Monads

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

[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

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 the 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. In our case, this is useful for encapsulating impure computations in a data structure that appears from the outside to be pure. This is explained in detail in Chapter 3.2.

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 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. His research is the cornerstone of this thesis.
Volume 4, Fascicle 5 of The Art of Computer Programming, released in November 2019 [Knu19], muses about a lot of exact cover problems and introduces an algorithm that is able to efficiently solve these problems, called *Algorithm X*. Algorithm X uses recursion and backtracking to find all possible solutions to an exact cover problem. This algorithm of his can be expanded to use **secondary items** and items that are **color-coded**. However, we will only expand it to be able to use the former, and not the latter.

3.1.1 Dancing Links

The main idea of Donald Knuth for this algorithm was to use **dancing links** as his technique to backtrack, hence why Algorithm X is also referred to as **DLX**. Dancing links is a technique for adding and removing a node from a circular doubly linked list while being able to do both in constant time.
A **doubly linked list** (DLL) 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.

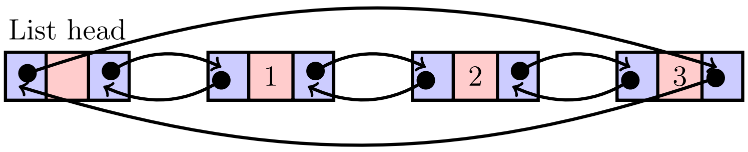


Figure 3.1: *Circular doubly linked list*

Dancing Links uses the linked structure of the circular doubly linked list to its advantage. The ideas are as follows:

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

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

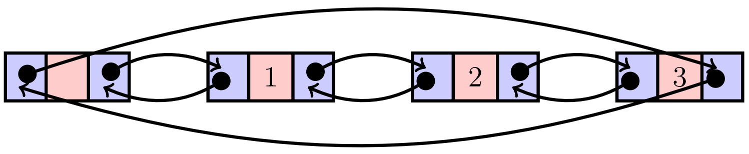
If x has to be restored (assuming that x.next and x.prev have been left unmodified), we do:

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

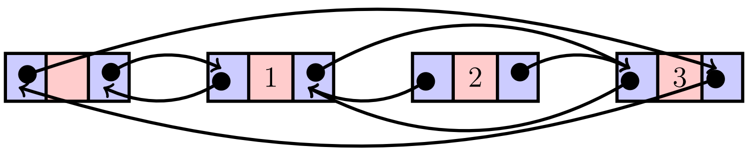
The adding and removing process of an item from every part of the doubly linked list is very fast (O(1)), especially if the item in question has to be restored after previously being removed from the list. This makes the data structure optimal for backtracking algorithms.

Why are the dancing links called like that, though? It is the choreography that underlines the motions of the pointers tangling and untangling themselves. Let us look at an example of dancing links in process:

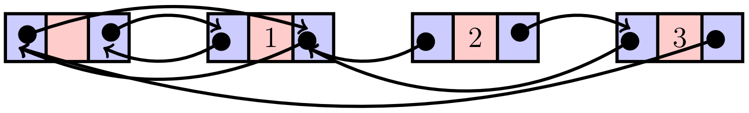
We start with an 3-element untangled list (the first element is the header element of the list):



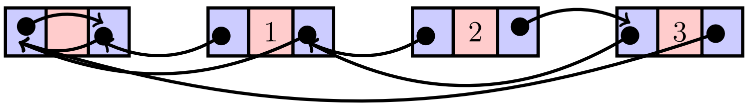
If we use (3.1) to delete element "2", the list becomes



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 now empty since the header element points to itself with both pointers. As we see, the links have become rather tangled, and trying to untangle them only knowing the last state of the list would be quite hard. However, because we know the order in which we deleted the elements (2 → 3

$\to 1$), 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 exact cover problems. 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 we need a constraint that has to be covered *at most once.* These elements are called **secondary items**
- The subsets of X that represent our available choices are called **options**, while the elements of X are called **items**

DLX is able to find solutions to a problem by *covering items.* Covering an item i means that we delete it from the item list (meaning that we already possess the item in our preliminary solution), and we delete all the options that contain this item. Thus, we have such a basic algorithm:

1. Select an item i that needs to be covered; terminate successfully if none are left (solution found)
2. If no active options involve i, backtrack or terminate unsuccessfully (there is no solution). Otherwise, cover item i.
3. 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.

After eliminating all possibilities, we want to backtrack to a previous state of the data structure. For that, we want to uncover the previously covered item. This is done by restoring all options that contain i as well as the item i in the item list.

Data Structure

We know the technique for backtracking and the basic procedure for DLX. However, what kind of data structure do we want to use for storing our items and options? Since we are going to be adding and deleting options and items a lot, we want to use a data structure that can use dancing links as efficiently as possible. So we are not going to use a doubly linked list that uses pointers and references, because that would take quite a lot of memory. Instead of that, 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 its corresponding previous and next item.

We want to have two separate arrays that both use a doubly linked list to link the elements to one another. One DLL is going to be for the items and another one for the options, where each element of the option is also called a **node**. The item array will have a list head at the start, so that if all items are covered, we will be able to check that by testing if the list head points to itself/a secondary item (we do not need to cover secondary items).

```
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 reference to its corresponding **item** saved in *itm*. Each node has a pointer to the node that is directly above and below it (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 pointers of the nodes in the options never change. So we do not need the pointers.([Knu])

   ```
   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 do we know when the option ends when we traverse it?
   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 points to the start of the preceding option, while their down field points to the end of the following option. Thus, it is easy to traverse an option circularly, in either direction.

An example of how the data structure would look after its initialisation is seen in 3.2. The numbers represent the indices in the option array/item array. The first row of the option array is also the whole item array (with **item** as its data structure instead of **node**). However, while the item array has left and right pointers and a value for each cell, the cells in the first row of the option

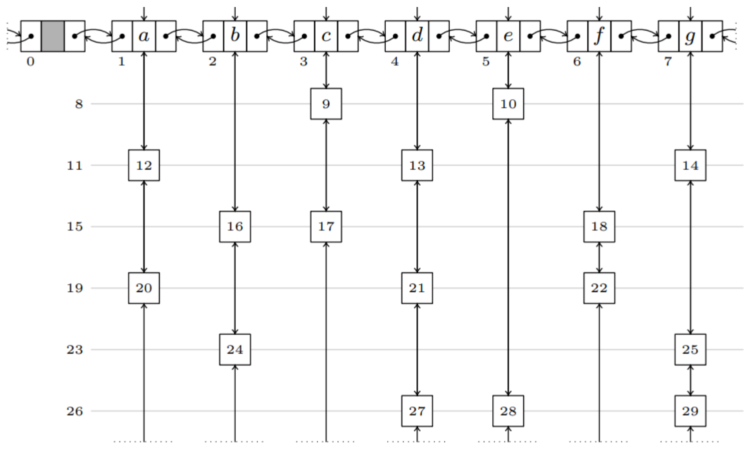


Figure 3.2: *Datastructure example for DLX [Knu]*

array have the first and last occurrence of the item in the node array. Its *itm* value is also not a reference to the **item** (because it is already on that index), but the number of uncovered nodes of that specific item. This is used for deciding which item should be covered next (with the MRV heuristic). While the topmost row of the node array represents all the items, the remaining rows represent the different options. The numbers to the left and right of the options are the spacer nodes. As an example, the *down* and *up* values of the spacer of index 11 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 [Knu19]):

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

In 3.3 for all options containing the item i the procedure *hide* is called. Afterwards, the item is removed from the doubly linked list.

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

hide(p) traverses the option that contains the node p and removes all its elements from the option DLL by using 3.1. Each time an element is removed, the length of that item is reduced by 1.

Uncovering an item is very similar to covering it. The only difference is that instead of removing an element, we recover it with 3.2. We also move in the opposite direction: In uncover, we move not downward but upward, and in unhide we move left instead of right.

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

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

Additionally, to the cover and uncover procedures, we need 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 [Knu19]

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 $x_0 x_1 \cdots x_{l-1}$ and go to X8.

X3. [Choose i.] At this point the item $i_1, \cdots, i_l$ still need to be covered, where $i_1 = \text{right}(0)$, $i_{j+1} = \text{right}(i_j)$, $\text{right}(i_t) = 0$. 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 $i \leftarrow \text{top}(x_l)$, $x_l \leftarrow \text{down}(x_l)$, 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. However, if we want it to be able to deal with secondary items, we need to slightly change the algorithm.
We 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 that need to be covered. Then we just need to be careful that the MRV heuristic ignores the secondary items. Afterwards, 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 only be 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 efficiently. The problem with this approach is that I/O is not referentially transparent, and the end product 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 was written at the University of Glasgow on March 10, 1994. Introduces **State Threads** which allow users to create stateful applications 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 referentially 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:

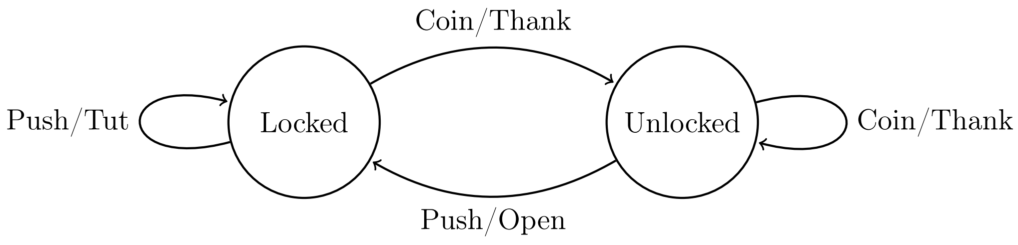


Figure 3.3: *Turnstile with States*

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 may change (but not necessarily, as we can see with the *Coin* input).

Stateful computations are also used to manipulate updatable states. This means that they can also manipulate mutable values, like an array. This is useful since we need an array for the DLX algorithm. We will be able to change the array without needing to recreate it every time from scratch. For this, *State Transformers* are going to be used, which are mutable like I/O but would not make all the functions using them suddenly impure.

State Transformers

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

- **[Complete referential transparency]** How is this possible, then, since state relies by definition on an I/O interaction to work? This works because a stateful computation is made in a *state transformer*. This is a function that takes the computation 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 only need 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**. One can think of it like this:

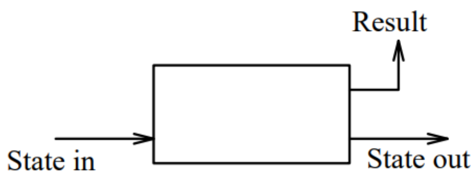


Figure 3.4: *State Transformer [LJ]*

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 for 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:

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

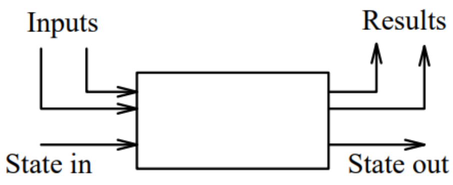


Figure 3.5: *State Transformer multiple Inputs and Outputs [LJ]*

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

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

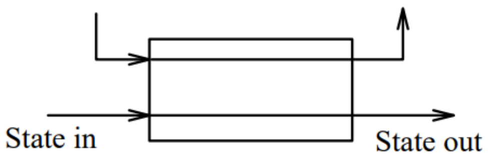


Figure 3.6: *returnST [LJ]*

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

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

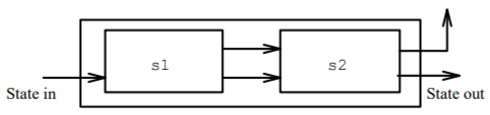


Figure 3.7: *thenST [LJ]*

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 as a part of a larger program that does not manipulate any state. 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 one. If this type did not exist, then the outputs would not be deterministic and could allow for different outputs depending on which thread went 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 a time. However, for our purposes, we want the state transformer to manipulate an array that can be thought of as multiple items, where each of them is independently mutable. Such an array can be created and 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** algorithm efficiently.

4 Implementation DLX

This part of the thesis will be about the implementation of Donald Knuth's DLX algorithm [Knu19].

4.1 DLX

Here, we are going to implement **DLX** into Haskell. Originally, Donald Knuth created DLX in C. As C is an imperative language and Haskell is functional, we need to make some changes (mostly we just need to work around the limitation of not being able to use global variables). The original implementation can be found in [**knuthwebiste**]. Because of the **State Transformer** we do not need to revamp the whole data structure and can use the array as described in the original implementation.

4.1.1 Doubly Linked Lists

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

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

The **node array** will contain the data structure Node:

```
data Node a = 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 items with his pointers but would not have a name. In C, we can leave it uninitialized. However, in Haskell, we cannot do that, so we need to give it a value that can be identified as invalid. 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 **Algorithm 2**. The if-case

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
```

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

The procedure for the **node array** is quite a bit harder, since we have to connect all the nodes that share the same item name, as well as link the spacer nodes to the first node of the previous option and the 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, after creating them between options, we 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 **Algorithm 3**.

4.1.2 Algorithm

The DLX algorithm is described in Chapter 3.1 in **Algorithm 1** and can be implemented mostly as described. The only problem is that we cannot divide the algorithm into parts as easily in **Haskell** as in C. That is because we do not have access to global variables and have a static type system.

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 only one index of one of its elements) that produces an exact cover to the problem.

Since the options in the solution are only made up of one index of one of the options elements, we implement a quality of life function that finds all other elements of the options and converts the indices of the elements to their names. This is done in **Algorithm 4**

Algorithm 3 Create Node STArray

```
Input: options, 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 options elems to $l$. Set $itm = i$, where $i$ is the index of the
   item in itemArr that matches the options elem. 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 $l$
```

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
        $j$ ← find start of option $i$ ▷ index of $left\ spacer\ node + 1$)
        $opt$ ← 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, we have not shown how the pseudocode looks when

it is done.
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 should not cover each item but only the primary ones.

```
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 non-primary items by filtering out the items that have an index higher or equal to n.

4.2 Finding solutions with DLX

Here we are going to explore the established way to get to a solution to a problem with the DLX algorithm.

4.2.1 Basic Examples

Let us look at some basic examples for the DLX to solve.
The real-world example from 1.1 is a good first test run for the DLX algorithm, as we already know the solution. We got the **primary items** *A, B, C, D, E, F* which will be input as a list of strings to DLX. The **secondary item** list is going to be empty, as there are no secondary constraints in this example.
Next, we have the **options**:

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

Each option is going to be a list of Strings. The elements are going to be Strings. If we put the defined **options** and **items** into DLX, we 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, we will use the items *A, B, and C*, where *A, B* are the **primary items**, and *C* will be a **secondary item**. The **options** are:

A, B
A, B, C

If we use this input for the algorithm, we get two solutions:

Solution A	Solution B
A B	A B C

Table 4.1: *Problem with secondary Item*

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

4.2.2 Queens

The Queens problem was already talked about in Chapter 2.1. In summary, we want to know all solutions to the problem of placing n queens on a n×n board without them attacking each other. As they can move horizontally and vertically, all the queens have to be in 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** in the solution. These are the **primary items**. For the **secondary items** we have the **upward diagonals and downward diagonals** as they are present more than n times on the board. Since all queens can move diagonally as well, this constraint is necessary. As such, we have the row and column primary constraints and upward and downward diagonal secondary constraints.

We want to generalize the problem to **n** Queens. As such, we need an algorithm that can create different **DLX formats** depending on the **n**. The constraints are defined in 2.1 (row constraint), 2.2 (column constraint), 2.3 (upward diagonal constraint), and 2.4 (downward diagonal constraint). The procedure for **constructing the Format** looks like this:

Algorithm 5 Queen DLX [Knu]

```
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 $1 \cdots n$ while adding each
   number to the string "r" and "c"
3. Construct the secondary items by iterating from $2 \cdots (2 \cdot n)$ 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.
```

The downward constraint is changed slightly and adjusted to the upward constraint so that both share the same numbers.

Now we can run the procedure for every natural number n, and a format will be created that DLX can understand and find solutions to.
Since, the output by DLX is in the format that we gave it, it would be better to convert that format back to a more intuitive way of understanding the solution (aka a board state). 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*

However, 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, we are going to generalize this *interpret* algorithm so that it can convert the solutions to many problems into a more understandable 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 (also called **boxes**) contain all the digits 1 through 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 an integer. The top left box is labeled as 0, the one to the right as 1, the next one as 2, etc., until we have 8 at the bottom right.

The idea for the algorithm is that we have different items for the positions (of the cells), rows, columns, and boxes. Every option has to contain one position, one row, one column, and one box item. They 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 the *inputValue* in each option. Options are created only for empty cells. After figuring out what values for the row, column, and box that the empty cell is in are missing, the algorithm makes each combination where the *inputValues* of row, column, and box match. Since there might be more than one option for each cell, we need the position item so that only one option for each cell is selected.

Algorithm 7 Create Sudoku DLX format [Knu]

```
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 Queens problem, we want the solution to be more readable. So we 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, for example, be 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 was to create an algorithm that produced a format of the problem that could be input into the DLX algorithm. This is quite a hindrance for people not knowing the programming language or even having no programming experience. Hence, we want to create a DSL that makes solving exact cover problems easy and intuitive.

5.1 Data Structures

The basic idea 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 some shape. The creation of the format and the inputs differ greatly from, for example, **Sudoku**. That is why we first want to generalize all **shape-based problems** like **Queens** and **Sudoku** before generalizing them even more.

5.1.1 Inputs

Our data structure has to be able to take in different possible **Inputs** for the problem. For Queens, it might be *Q* while for the standard Sudoku, it would be 9 inputs, *1* through *9*. There are some problems where an input has to occur multiple times per shape. An example of that is the **Ian** problem, where the input *1* occurs once per row/column, *2* occurs twice, and so on until *4*. So that such problems can be solved, we need a way to specify the occurrence amount.

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

Input has two expressions

- **InputsAndOccurences** takes a list of tuples as its argument. The tuples consist of the number the input occurs in a shape, and the input's name.
- **ManuallySelectOccurence** takes a list of input names as its argument. After the code starts running, it would ask us to manually input how often an input occurs for each constraint (notice that **InputsAndOccurences** defines it for all constraints; as such, this option is more flexible).

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

$$\textbf{occurs}\ \text{input i} = [(\text{i, input})] \tag{5.1}$$

$$\textbf{occurEach}\ \text{inputs i} = \text{map (i,) inputs} \tag{5.2}$$

$$\textbf{next}\ \text{a b} = \text{a ++ b} \tag{5.3}$$

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 solution to the problem. Even though there are exact cover problems that use three-dimensional boards or even higher-dimensional ones, we restrict ourselves to a board size of two dimensions for now.

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

The user inputs the initial state of the board into **Board**. If the initial board is an empty rectangle, like in Queens, we can use the function:

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

Since we can set the initial state of the board, we can now restrict a problem's output by fixing an input in the initial board state. If we take 4 Queens and restrict the first input to

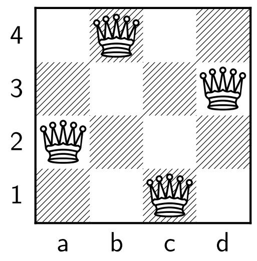
Board		**Solution**
[[""],["Q"],[""],[""], [""],[""],[""],[""], [""],[""],[""],[""], [""],[""],[""],[""]]	$\Longrightarrow$	

Table 5.1: *Restricted Solutions for 4 Queens Problem*

We get only one solution.

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 define his own custom constraints for some problem that his **Inputs** and **Board** have to satisfy. It is defined as

$$\textbf{data Constraints} = \text{Constraints [Int]}$$

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:

$$\begin{aligned}
&\textbf{rowCons} = [1]\\
&\textbf{colCons} = [2]\\
&\textbf{downDiagCons} = [3]\\
&\textbf{upDiagCons} = [4]\\
&\textbf{boxCons} = [5]\\
&\textbf{customCons} = [6]
\end{aligned}$$

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 custom shape.

5.1.4 Exact Cover

Putting it all together, we get

```
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.

Let us say we want to solve the **8 Queens problem**. We would define the data structure as follows:

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

5.2 DLX Formating

Now, how do we get from an **Exact_Cover** instance to a solution? Since we want to use DLX as the solver, we need to create a **DLX Format** that can be recognized by the DLX algorithm.

The basic idea is to go from constraint to constraint, creating for each one its own format, and combining the same-cell ones with each other (only the items with the same input are put together). 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 DSL can, however, be expanded to contain 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 occurence 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 for each constraint.

Next, we have a function that creates an item for each cell on the board and puts each in a separate list of a list of a list. 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 sublist 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 sublist/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 sublist 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 *transfromedBoard[i]*; **If** *elem* is empty || *itemDigits[i] - (ItemDigits[i+1]-1)* < 0 goto 6. **Else** goto 7.
6. **Designate** field as invalid
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* (using **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**. Step 6 might be a bit confusing. The field is marked as invalid because if no options for one constraint can be constructed at that cell, then no options for other constraints will be able to be combined into a format on that cell (they need the same input for that). This can be observed in **Algorithm 11**.

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 into one.

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 primary constraints
secondary items ← Remaining items
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 to occur exactly/at most their *occurence rate* in a given shape**

5.2.2 Transform functions

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

- **[rowCons]** is the most basic one. We do not have to make any changes to the baseFormat/board, as each row is already in a separate list.
- **[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 funcion *diagonals*
- **[boxCons]** we use two nested for loops, where the first for loop goes row wise and the second one goes 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 this). 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 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 needs 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 will get a format with which we are able to solve the problem.

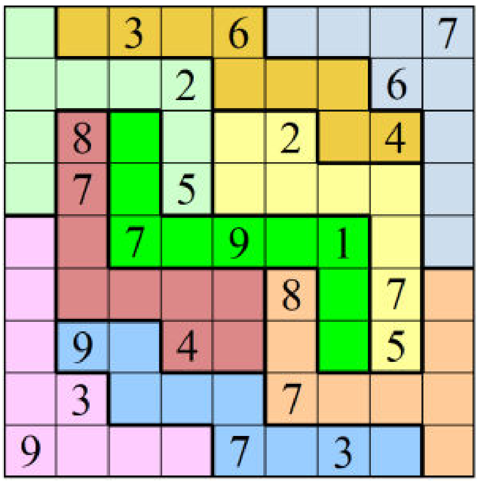


Figure 5.1: *Jigsaw Sudoku*

5.2.3 Quality of life

When we use the generated 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 with 2 or more constraints looks 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 solutions 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 extend it with *showXSolutions :: Int* (shows only the first x solutions), *interpret :: Bool* (asks if we want to use **Algorithm 13**), and so on.

5.3 Extensions for DSL

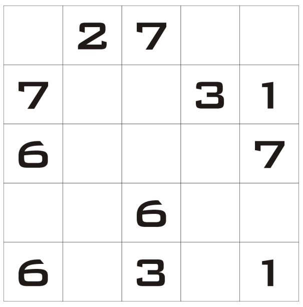
We mentioned previously that our DSL is pretty limited in what problems it is able to solve (namely, only shape-based problems). It is of course possible to extend the constraints that our DSL is able to work with, but we cannot use **Algorithms 10, 11,** and **12** for that. New methods are needed to create the format (naturally, it would be best if the methods were 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

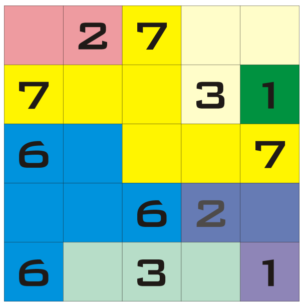
Fillomino is a logic puzzle that is played on a rectangular grid with no standard size. Some cells of the grid start with already-existing numbers, called **givens**. The goal is to divide the grid into regions called **polyominoes** such that each given number n in the grid satisfies the following constraints:

- 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 givens 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 [Knu], 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-items (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, the topic of solving exact cover problems efficiently and intuitively is discussed. To do that, we first needed to know how an exact cover problem is defined. As such, we introduced the types of constraints that a problem might have. These are the *primary constraints* and *secondary constrainsts*. **Primary constraints** are the ones that need to be covered *exactly once*, while **secondary constraints** need to be covered *at most once*. Next, the options were introduced. They are the subsets of constraints (also known as items) that the solver can choose to include in its solution, which hopefully will satisfy all constraints. The state-of-the-art **Algorithm X**, better known as **DLX**, is shown to be such a solver. It checks if all primary constraints are covered and backtracks if no options are available that would satisfy more constraints without covering an already satisfied constraint. The backtracking is done via **dancing links**. This technique deletes and recovers elements in constant time while preserving the structure of the data.

The 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 the shape of the board. This is done by reducing the constraints into a row-shaped constraint by transforming the board with an appropriate function. A problem is also shown that is not solvable by the described DSL. However, a simple way to extend the language is presented to also capture the necessary constraints. With this, the problem can also 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. Since the DSL in its current form is not that powerful, further research can be made to extend it by generalizing more constraints or, if possible, all constraints. Another avenue of research can be 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

Bibliography

[Knu] Donald Knuth. *Programs to Read*. URL: `https://www-cs-faculty.stanford.edu/~knuth/programs.html`. (accessed: 28.11.2023).

[Knu19] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascile 5*. Addison-Wesley, 2019.

[LJ] John Launchbury and Simon L Peyton Jones. *Lazy Functional State Threads*. URL: `https://www.microsoft.com/en-us/research/wp-content/uploads/1994/06/lazy-functional-state-threads.pdf`.