Floyd and Warshall meet Kleene

Marian Haselrieder

Bachelor Thesis

Supervisor:
Dr. Vincent van Oostrom
Department of Computer Science
Universität Innsbruck

Innsbruck, September 10, 2020

Abstract

Warshall's algorithm computes the transitive closure of a binary relation, Floyd's algorithm calculates the shortest path between all pairs of nodes in a weighted multigraph and Kleene's algorithm converts a nondeterministic finite automaton with ϵ-moves into a regular expression. Three different algorithms solving three different problems. A closer look shows, that they don't only share many characteristics, but are almost identical. This bachelor thesis shows, that all these algorithms are instances of the same problem related to Kleene algebras. It also presents a general solution for all Kleene algebras, such that these three and other similar problems can be solved using one and the same algorithm. Furthermore a tool that has been developed as part of this project is presented: It shows how the general algorithm can be used to solve all mentioned problems and highlights their similarities, which is especially useful when teaching these topics.

Contents

1 — Introduction

Warshall's algorithm takes the adjacency matrix A of a binary relation R as an input and replaces it with the adjacency matrix of its transitive closure R^+. Floyd's algorithm also takes a matrix A as an input, an adjacency matrix representing a weighted multigraph G. In this case A is replaced with a matrix containing the weights of the shortest paths between al pairs of nodes. Kleene's algorithm takes a matrix representing a nondeterministic finite automaton as an input. It then computes one regular expression for each pair of nodes, which represents the accepted language going from the first node of the pair to the second one.

All three algorithms eliminate or add one node after another (depending on the point of view). Whenever adding a node n_k the algorithms check, whether an entry $A_{i,j}$ containing a value for the pair of nodes (n_i, n_j) can be replaced or extended by a path now also going through n_k. This boils down to generating all possible paths through a graph or automaton, where each algorithm performs different operations on the paths:

- Warshall's algorithm only needs to find one path for a pair of nodes, meaning that the transitive closure of the relation contains this pair.
- Floyd's algorithm chooses the smallest of the paths for each pair of nodes.
- Kleene's algorithm keeps all generated paths where each edge/transition has it's own symbol and hence generates a regular expression for the accepted language going from one node to another.

These similarities of the algorithms can and will be generalized in the following chapters. Firstly a detailed explanation of the different algorithms and their properties will be given. Then Kleene algebras will be introduced, the structure to which all these algorithms comply to. Afterwards a general solution for the problem will be presented as well as a web application showing a step-by-step solution and the similarties of all three algorithms.

2 — Related work

Within the scope of this bachelor thesis project, only the algorithms of Floyd, Kleene and Warshall will be discussed. But there also exist other algorithms and problems which comply to the same pattern, for example *State Elimination* [11].

There are several open problems regarding Kleene's algorithm or equivalent algorithms on Kleene algebras. One would be that an exponential growth of the length of regular expressions or algebraic terms is sometimes inevitable. This topic will shortly be discussed as part of this thesis, but more details can be found in the papers *Simplifying Regular Expressions, A Quantitive Perspective* [6] and *Representation of Events in Nerve Nets and Finite Automata* [9]. More information regarding the time complexity of the simplifications can be found in the paper *The minimum consistent DFA problem cannot be approximated within any polynomial* [14].

3 — Generalization of the algorithms

1 The three Algorithms

In this section we look at all three algorithms in a more detailed manner.

1.1 Floyd's Algorithm

Floyd's algorithm [13] computes the shortest path between all pairs of vertices in a weighted finite multigraph G. Both directed and undirected multigraphs are allowed. Theoretically also negative edge weights are possible, with the restriction that there aren't any negative cycles.

Let G be a multigraph with the nodes $n_0, n_1, n_2, \ldots$ and the edges $e_0, e_1, e_2, \ldots$ where $w(e_i)$ denotes the weight of the edge e_i. We construct the matrix A as follows:

$$A_{[i,j]} = \begin{cases} 0, & \text{if } i = j. \\ \infty, & \text{if there is no edge from } n_i \text{ to } n_j \\ min(\{b(e_k) \mid e_k \text{ is an edge from } n_i \text{ to } n_j\}), & \text{otherwise} \end{cases}$$

Then the following algorithm overrides the matrix A with a new matrix, such that $A_{[i,j]}$ contains the length of the shortest path from the node n_i to the node n_j:

Algorithm 1: Floyd's Algorithm

```
$len \leftarrow$ length of $A$
for $k \leftarrow 0 \ldots (len - 1)$ do
    $B \leftarrow A$
    for $i \leftarrow 0 \ldots (len - 1)$ do
        for $j \leftarrow 0 \ldots (len - 1)$ do
            $B_{[i,j]} \leftarrow min(A_{[i,k]} + A_{[k,j]}, B_{[i,j]})$
    $A \leftarrow B$
```

The algorithm bases on the following perception: If the shortest path from e_a to e_b goes through the node e_c, the path from e_a to e_c and the path from e_c to e_b must already be minimal. Let's assume we already know the shortest paths between all pairs of nodes with an index less than k. To calculate the shortest paths between all pairs of nodes with an index less or equal to k there are two options for a path from e_i to e_j. Either the already known path going only through nodes with index smaller than k from e_i to e_j stays the shortest, or the known paths going from e_i to e_k and then from e_k to e_j is even shorter. The algorithm starts with $k = 0$ and increases k constantly after updating all entries. The proof of the algorithm is being omitted here, since it just formalizes this concept.

Previously we noted, that graphs containing negative cycles are not allowed by the algorithm. It's obvious that a graph containing a negative cycle contains paths of length $-\infty$, when looping through the cycle infinitely. The algorithm does not detect the cylces at runtime and hence delivers wrong results. Nevertheless it can detect the cycles at the end of its execution, namely if at least one entry $A_{[i,j]}$ with $i = j$ contains a negative result. Although the algorithm can be used to detect negative cycles, there exist faster algorithms as the *Bellman–Ford algorithm* [4].

To demonstrate how the algorithm works, a step by step solution is presented in Figure 3.2. It performs the algorithm on the multigraph shown in Figure 3.1, starting from the created matrix A and increasing the variable k step by step:

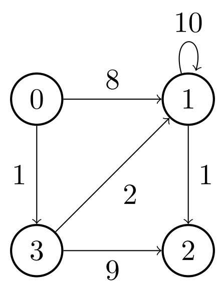


Figure 3.1

$$\begin{pmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ \infty & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{pmatrix} \xrightarrow{A_1} \begin{pmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ \infty & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{pmatrix} \xrightarrow{A_2} \begin{pmatrix} 0 & 8 & 9 & 1 \\ \infty & 0 & 1 & \infty \\ \infty & \infty & 0 & \infty \\ \infty & 2 & 3 & 0 \end{pmatrix} \xrightarrow{A_3}$$

$$\begin{pmatrix} 0 & 8 & 9 & 1 \\ \infty & 0 & 1 & \infty \\ \infty & \infty & 0 & \infty \\ \infty & 2 & 3 & 0 \end{pmatrix} \xrightarrow{A_4} \begin{pmatrix} 0 & 3 & 4 & 1 \\ \infty & 0 & 1 & \infty \\ \infty & \infty & 0 & \infty \\ \infty & 2 & 3 & 0 \end{pmatrix}$$

Figure 3.2

1.2 Warshall's Algorithm

Warshall's algorithm [13] takes a binary relation as an input and calculates its transitive closure, the smallest transitive relation on X that contains R. The procedure works as follows:

Let A be the adjacency matrix of some binary relation R on a set X, then the following algorithm overrides A with the adjacency matrix of R^+, the transitive closure of the relation R:

Algorithm 2: Warshall's Algorithm

```
$len \leftarrow$ length of $A$
for $k \leftarrow 0 \ldots (len - 1)$ do
    $B \leftarrow A$
    for $i \leftarrow 0 \ldots (len - 1)$ do
        for $j \leftarrow 0 \ldots (len - 1)$ do
            if $A_{[i,k]} = 1$ and $A_{[k,j]} = 1$ then $A_{[i,j]} \leftarrow 1$
            else $B_{[i,j]} \leftarrow A_{[i,j]}$
    $A \leftarrow B$
```

Take the relation $R = \{(0,1), (0,3), (1,1), (1,2), (1,3)\}$ on the set $X = \{0,1,2,3\}$ as an example. The respective graph representation and adjacency matrix of the relation would be the following:

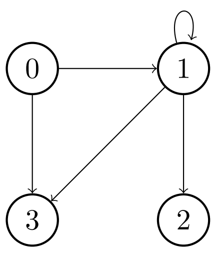


Figure 3.3

$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 3.4

For this input Warshall's algorithm computes the set $R^+ = R \cup \{(0,2)\}$, since $(0,2)$ is the only pair that has to be added, to make R transitive.

It is easy to see, that Warshalls's algorithm is quite similar to Floyd's algorithm. Especially when looking at the graph representation of the relation it becomes clear, that the transitivity of a relation is equivalent to the reachability within its graph. Note that the definition of reachability requires at least one edge between two nodes and hence not all pairs (a,a) are being added automatically.

Warshall's algorithm can easily be changed to compute the reflexive transitive closure of a relation, by modifying the matrix construction. It is sufficient to set the entries $A_{[i,i]}$ of the adjacency matrix to 1 for all $i = 0, 1, \ldots, |X|$. One could also modifiy the algorithm and matrix construction to use a third value, which denotes pairs that have not been added because of transitivity but because of the reflexive property. Using this approach one could extract both the transitive and the reflexive transitive closure from the resulting matrix.

1.3 Kleene's Algorithm

Kleene's algorithm [7] converts a nondeterministic finite automaton with ϵ-moves into a regular expression that represents the language accepted by the automaton. More intuitively: It returns a regular expression that represents all words than can be formed by beginning at a starting state, pathing through the automaton and stopping at an accepting state.

Let G be an NFA-ϵ with states $s_1, s_2, \ldots$ and edges $e_1, e_2, \ldots$ where $w(e_i) = a$ denotes the symbol a of the edge e_i with $a \in \Sigma \cup \{\epsilon\}$. Then, just like Floyd's algorithm, a matrix has to be generated from the input:

$$A_{[i,j]} = \begin{cases} \epsilon + w(e_u) + \ldots + w(e_v), & \text{if } i = j \text{, with } e_u, \ldots, e_v \text{ being} \\ & \text{all the edges from } s_i \text{ to } s_j \\ \varnothing + w(e_u) + \ldots + w(e_v), & \text{otherwise, with } e_u, \ldots, e_v \text{ being} \\ & \text{all the edges from } s_i \text{ to } s_j \end{cases}$$

The reason we have ϵ in addition to all transitions for $i = j$ is that one can "loop" or "remain" in a state by doing ϵ-moves without changing the resulting word. In contrast to that we set $A_{[i,j]}$ to $\varnothing$ if $i \neq j$ and no ϵ-move or other transition from s_i to s_j is available. Note that $\varnothing + a = a$ and hence the $\varnothing$ simplifies away if any transition exists.

Kleene's algorithm takes this matrix A as an input and replaces it with a matrix containing a regular expression for each pair of states. Each regular expression represents the language accepted by the automaton by moving from the first state of the pair to the second one.

Algorithm 3: Kleene's Algorithm

```
$len \leftarrow$ length of $A$
for $k \leftarrow 0 \ldots (len - 1)$ do
    $B \leftarrow A$
    for $i \leftarrow 0 \ldots (len - 1)$ do
        for $j \leftarrow 0 \ldots (len - 1)$ do
            $B_{[i,j]} \leftarrow A_{[i,j]} + A_{[i,k]}(A_{[k,k]})^* A_{[k,j]}$
    $A \leftarrow B$
```

Since an automaton has explicit starting an accepting steps, only the entries of the pairs (s_i, s_j) where s_i is a starting state and s_j is an accepting state are relevant for the solution. To form one solution out of these entries, they are being concatenated with the + operator.

The automaton displayed in Figure 3.5 with its corresponding matrix in figure 3.6 serves as an example. After performing the algorithm and simplyfing all the entries (more about this in section 4.6) one ends up with the matrix in Figure 3.7. As only $A_{[0,1]}$ is relevant for the solution, one ends up with the regular expression $a*b(a*b)*$.

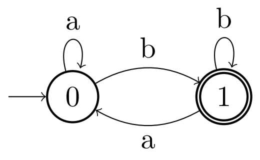


Figure 3.5

$$\begin{pmatrix} \epsilon + a & \varnothing + b \\ \varnothing + a & \epsilon + b \end{pmatrix}$$

Figure 3.6

$$\begin{pmatrix} (a + bb*a)* & a*b(a*b)* \\ b*a(b*a)* & (b + aa*b)* \end{pmatrix}$$

Figure 3.7

Proof

In this section we prove that Kleene's algorithm truly returns the mentioned result. It is sufficient to prove the following invariant by performing induction on k: $A_{k[i,j]}$ (denoting $A_{[i,j]}$ after the k-th iteration) contains the regular expression representing all words that can be formed by moving from s_i to s_j only visiting states with an index smaller or equal to k.

Base case $k = 0$:
Follows directly from the construction of $A_{0[i,j]}$.

Step case $k - 1 \to k$:
Assuming there is a path p from s_i to s_j only visiting states with an index smaller or equal to k we can distinguish two cases:

- p doesn't visit k: Then we can use $A_{k-1[i,j]}$ to describe the corresponding words.
- p visits k: Then we can split the p into multiple pieces. The first one is a path from s_i to s_k that doesn't visit s_k, the last one is a path from s_k to s_j that doesn't visit s_k. Inbetween there is an arbitrary amount of paths from s_k to s_k that are not visiting s_k. It follows from the I.H. that the regular expression $A_{k-1[i,k]}(A_{k-1[k,k]})*A_{k-1[k,j]}$ represents the set of all words that can be formed.

Combining the two regular expressions we get $A_{k-1[i,j]} + A_{k-1[i,k]}(A_{k-1[k,k]})*A_{k-1[k,j]}$ which represents all words that can be formed by moving from s_i to s_j only visiting states with an index smaller or equal to k. Since the algorithm assigns exactly this term to $A_{k[i,j]}$, the invariant holds. □

1.4 Similarities and differences

Code and structure

Not only are all three algorithms dynamic programming algorithms, but they also only differ by a single line of code (without taking the matrix creation into consideration). They all update a matrix several times based on its previous values and they all iterate through the nodes or states in the same order. Each algorithm introduces an alternative between two values, where the first one is the value from the previous iteration and the second one is computed by taking into consideration the same new node as an "intermediate step". The algorithms slighty differ in the creation of the matrix, more about this in section 4.1.

Complexity

It's trivial to see that all three algorithms run in $\mathcal{O}(n^3)$ where n denotes the total number of nodes or states. Also regarding the space complexity there are hardly any differences. All three store $\mathcal{O}(n^2)$ values, one for each pair of nodes or states. It's important to point out though, that Kleene's algorithm takes up more than $\mathcal{O}(n^2)$ space, since the length of the regular expressions grows exponentially [5]. This topic will be picked up in section 4.6.

Graph interpretation of the algorithms

Thinking of the algorithms as graph problems highlights their similarities even more. One could think of Kleene's algorithm as an algorithm generating all possible paths between every pair of nodes. Floyd's algorithm does the same, but whenever a new path is being found it doesn't add it as an alternative but throws away the longer path since it doesn't contain any relevant information. Also Warshall's algorithm sort of computes all possible paths between every pair of nodes, but after finding one suitable path for a pair of nodes it throws away all additional information since it isn't relevant anymore. So all three algorithms perform the same "path generation", but hold on to different information. One could generally say that for every algorithm all edges and paths have a certain value and each algorithm has their own operation on choosing between two paths/values, concatenating two paths/values or iterating a path/value. There is indeed a more general structure which generalizes this concept, the Kleene algebra. In the next section a definition of Kleene algebras will be given as well as a suitable algebra for each of the algorithms.

2 Kleene Algebra

2.1 Definition

A Kleene algebra is an algebra $(A, +, \cdot, 0, 1, *)$ where the operations $+ : A \times A \to A$, $\cdot : A \times A \to A$ and $* : A \to A$, satisfy the following axioms:

$(A, +, \cdot, 0, 1)$ is a semiring:

(1) $a + 0 = 0 + a = a$ (Identity element of $+$)
(2) $(a + b) + c = a + (b + c)$ (Associativity of $+$)
(3) $a + b = b + a$ (Commutativity of $+$)
(4) $a \cdot 1 = 1 \cdot a = a$ (Identity element of $\cdot$)
(5) $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ (Associativity of $\cdot$)
(6) $a \cdot 0 = 0 \cdot a = 0$ (Annihilation by 0)
(7) $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ ($\cdot$ left distributes over $+$)
(8) $(a + b) \cdot c = (a \cdot c) + (b \cdot c)$ ($\cdot$ right distributes over $+$)

$+$ is idempotent:

(9) $a + a = a$

$*$ has the following properties where the partial order $\leq$ on A is defined as: $a \leq b \iff a + b = b$:

(10) $1 + a \cdot a* \leq a*$
(11) $1 + a* \cdot a \leq a*$
(12) $b + a \cdot x \leq x \Rightarrow a* \cdot b \leq x$
(13) $b + x \cdot a \leq x \Rightarrow b \cdot a* \leq x$

2.2 Interpretation and further information

When working with Kleene algebras, each operation and identity element has a certain and similar role. The table 3.8 should give the reader an idea of what the meaning of each operation and identity element usually is:

Operation	Intuition
$+$	Choice, union, addition
$\cdot$	Sequential composition, multiplication
0	Neutral element of $+$, fail, false
1	Neutral element of $\cdot$, skip, true
$*$	Iteration, $a* = 1 + a + aa + \ldots$

Figure 3.8

An example of a Kleene algebra, as one can easily verify, would be the following (note that it is equivalent to the algebra proposed in section 2.3 for Warshall's algorithm):

$$(\{\mathit{true},\ \mathit{false}\}, \vee, \wedge, \mathit{false}, \mathit{true}, *) \text{ with } a* = \mathit{true} \text{ for all } a.$$

The priorities of the operators are the following: The * operator has a higher priority than the $\cdot$ operation and the $\cdot$ operator has a higher priority than the + operator. Instead of $a \cdot b$ one can simply write ab.

There are multiple lemmas and equivalences regarding Kleene algebras, which can be used to simplify elements and expressions. Here we only list a few of them, namely the ones that have been used in our tool to simplify elements of the algebra (together with the axioms (1), (4) and (6) 2.1). More about this in section 4.6.

a) $0* = 1$

b) $1* = 1$

c) $(a*)* = a*$

d) $(1 + a)* = (a + 1)* = a*$

2.3 The three Kleene algebras

In this section we provide a Kleene algebra for each of the three algorithms, which can later be used with the general algorithm.

Kleene algebra for Warshall's algorithm

Finding a Kleene algebra fitting the needs of Warshall's algorithm is quite intuitive. It's sufficient to use the two elements 1 and 0 as truth values: 1 denotes that a pair is part of the relation, 0 that it isn't. Hence the + operation corresponds to a logical $\vee$, meaning that if some elements demand a pair to be added (and some others might not), we still have to add it to make the relation transitive. The $\cdot$ operation corresponds to a logical $\wedge$, meaning that two pairs (a, b) and (b, c) only demand the pair (a, c) to be added to the relation, if both pairs are already part of the relation. Obviously 0 is the neutral element of $\vee$ and 1 the neutral element of $\wedge$. Since $a* = 1 + a + aa + \ldots$, the * operation trivially evaluates to 1 for both 0 and 1.
This results in the following algebra and operations:

$$(\{0, 1\}, \vee, \wedge, 0, 1, *)$$

$\vee$	0	1
0	0	1
1	1	1

$\wedge$	0	1
0	0	0
1	0	1

*	
0	1
1	1

Kleene algebra for Floyds's algorithm

A suitable set for the algebra would be $A = \mathbb{N}_0 \cup \{\infty\}$, where ∞ denotes that there is no path (which can also be seen as a path of infinite length). It is worth pointing out that variations of the sets $\mathbb{Z}, \mathbb{Q}$ or $\mathbb{R}$ could be used instead (by including $+\infty$ and $-\infty$ and adapting the following function definitions). The $+$ operation chooses the minimum of two elements and hence the shorter path of two available options. The $\cdot$ operation adds the length of two sequenced paths together. ∞ is the neutral element of $+$, since every path is shorter than a path of infinite length. 0 is the neutral element of $\cdot$, since it doesn't change the length of a path. Since the $*$ operation corresponds to $1 + a + aa + \dots$ where 1 denotes the element 0 (i.e. a path of length 0) it becomes clear that $a* = 0$ for all a. We end up with the following algebra:

$$(\mathbb{N}_0 \cup \{\infty\}, +, \cdot, \infty, 0, *)$$

$$a + b = min(a, b) \qquad a \cdot b = add(a, b) \qquad a* = 0$$

Kleene algebra for Kleene's algorithm

In this and the following sections, whenever both the operations of Kleene algebras and regular expressions are used, we always use $+^K, \cdot^K$ and $*^K$ for the operations of the Kleene algebra and $+^R, \cdot^R$ and $*^R$ for the operations of regular expressions.

Let Σ be some alphabet and $S = \{x \mid x \text{ is a regular expression over } \Sigma\}$ be the set of all regular expressions over Σ. Using S as the set of the Kleene algebra leads to multiple problems, one being that two different regular expressions defining the same language are not automatically equivalent regarding the algebra. Also one cannot simply take $+^R, \cdot^R$ and $*^R$ as operations of the algebra, since then $a +^K \varnothing = a +^R \varnothing$ which is not equivalent to a as the axioms of Kleene algebra demand. Therefore a different approach is being used. We say two regular expressions $a \in S$ and $b \in S$ are equivalent and write $a \equiv b$, whenever $L(a) = L(b)$. Clearly $\equiv$ is an equivalence relation on S where $[a]$ defined as $[a] := \{x \mid x \equiv a\}$ denotes the equivalence class of some $a \in S$ (the set of all regular expressions that define the same language as a). We define P as the set of all equivalence classes of $\equiv$ i.e. as S modulo $\equiv$ and it follows directly that P is a partition of S. Using P as the set of the algebra and hence the equivalence classes as elements solves the problem regarding the equality and results in simple operations, as they can be shifted to the respective regular expressions. Hence we intuitively get $[a] +^K [b] = [a +^R b]$, $[a] \cdot^K [b] = [a \cdot^R b]$ and $[a]*^K = [a*^R]$. The neutral element of $+^K$ is $[\varnothing]$ and the neutral element of $\cdot^K$ is $[\epsilon]$. It is easy to see, that these operations satisfy all the necessary axioms. We end up with with the following Kleene algebra and operations:

$$(P, +^K, \cdot^K, [\varnothing], [\epsilon], *^K)$$

$$[a] +^K [b] = [a +^R b] \qquad [a] \cdot^K [b] = [a \cdot^R b] \qquad [a]*^K = [a*^R]$$

Allthough the elements of the algebra are equivalence classes, it suffices to use one regular expression to represent an equivalence class for inputs, storage and return values. When we refer to the language of an equivalence class x (which is the language that all of its elements define), we informally also write $L(x)$.

3 Homomorphims

With the just defined algebras for the three algorithms, it is now possible to introduce homomorphisms to further emphasize their similarities and to introduce a hierarchy: Let K be the Kleene algebra for Kleene's algorithm, F the Kleene algebra for Floyd's algorithm and W the Kleene algebra for Warshall's algorithm. Recall that $L(x)$ is the language defined by the regular expression x and that $l(w)$ defines the length of a word w [12]. Furthermore, we define $min\_len(L)$ where L is a regular language as the minimum length of all words in L:

$$min\_len(L) = \begin{cases} 0, & \text{if } L = \varnothing \\ min\{l(w) \mid w \in L\} & \text{otherwise} \end{cases}$$

Lemma 1: $\varphi : K \to F$ defined as follows is a homomorphism:

$$\varphi(x) = \begin{cases} \infty, & \text{if } x = [\varnothing] \\ min\_len(L(x)), & \text{otherwise} \end{cases}$$

Proof: We have to show that $\varphi(a + b) = min(\varphi(a), \varphi(b))$ and $\varphi(a \cdot b) = add(\varphi(a), \varphi(b))$ hold. For the case that $a = \varnothing$ we get $\varphi(a+b) = \varphi(b) = min(\infty, \varphi(b)) = min(\varphi(a), \varphi(b))$ and $\varphi(a \cdot b) = \varphi(\varnothing) = \infty = add(\infty, \varphi(b)) = add(\varphi(a), \varphi(b))$. For $b = \varnothing$ the same reasoning can be done. Otherwise we have $\varphi(a+b) = min\_len(L(a+b)) = min\_len(L(a) + L(b)) = min\{l(w) \mid w \in L(a)+L(b)\}$ which is trivially equivalent to $min(min\{l(w) \mid w \in L(a)\}, min\{l(w) \mid w \in L(b)\}) = min(min\_len(L(a)), min\_len(L(b))) = min(\varphi(a), \varphi(b))$. Also $\varphi(a \cdot b) = min\{l(w) \mid w \in L(a)L(b)\}$ which is equivalent to $add(min\{l(w) \mid w \in L(a)\}, min\{l(w) \mid w \in L(b)\}) = add(\varphi(a), \varphi(b))$ since the concatenation of the shortest word of $L(a)$ with the shortest word of $L(b)$ is trivially the shortest word of $L(a)L(b)$. □

Lemma 2: $\psi : F \to W$ defined as follows is a homomorphism:

$$\psi(x) = \begin{cases} 0, & \text{if } x = \infty \\ 1, & \text{otherwise} \end{cases}$$

Proof: We have to show that $\psi(min(a, b)) = \psi(a) \vee \psi(b)$ and that $\psi(add(a, b)) =$

$\psi(a) \wedge \psi(b)$. If $a = \infty$ then $\psi(add(a, b)) = 0 = 0 \wedge \psi(b) = \psi(a) \wedge \psi(b)$ and $\psi(min(a, b)) = \psi(b) = 0 \vee \psi(b) = \psi(a) \vee \psi(b)$. The reasoning for $b = \infty$ is the same. Otherwise $\psi(min(a, b)) = 1 = \psi(a) \vee \psi(b)$ and $\psi(add(a, b)) = 1 = \psi(a) \wedge \psi(b)$. □

Lemma 3: The function $\phi : K \to W$ defined as follows is a homomorphism:

$$\phi(x) = \psi(\varphi(x))$$

Proof: The composition of two homomorphisms is a homomorphism. □

Lemma 4: $\chi : W \to F$ defined as follows is a homomorphism and an embedding:

$$\chi(x) = \begin{cases} \infty, & \text{if } x = 0 \\ 0, & \text{otherwise} \end{cases}$$

Proof: Omitted.

Lemma 5: $\tau : F \to K$ defined as follows is a homomorphism:

$$\tau(x) = \begin{cases} \varnothing, & \text{if } x = \infty \\ \epsilon, & \text{otherwise} \end{cases}$$

Proof: Omitted.

Lemma 6: The function $\kappa : W \to K$ defined as follows is a homomorphism:

$$\kappa(x) = \tau(\chi(x))$$

Proof: Omitted.

Clearly there is a homomorphism for each pair of the defined algebras, but the direction ($\varphi : K \to F$, $\psi : F \to W$, $\phi : K \to W$) is more interesting. As already mentioned in section 1.4, Kleene's algorithm retains more information than Floyd's algorithm and Floyd's algorithm retains more information than Warshall's algorithm. The same thing regarding the stepwise degradation of the information holds for the corresponding algebras and homomorphisms. The homomorphism $\varphi : K \to F$ shows that Kleene's algorithm can be used instead of Floyd's algorithms in case of all edges having the same weight (i.e. finding the path with the fewest edges). In the other direction one cannot compute any useful results using Floyd's algorithm instead of Kleene's. One can also see that both the elements of K and F, e.g. results from Floyd's or Kleene's algorithm, contain all the needed information to compute the solution for Warshall's alogrithm by using the homomorphisms $\psi : F \to W$ and $\phi : K \to W$. The homomorphism (and embeddings) for the other direction ($\chi : W \to F$, $\tau : F \to K$, $\kappa : W \to K$) are less imporant since they map from a less general to a more general structure. The fact that the functions are not isomophisms underlines this fact even more. All this shows the structural similarities and the hierarchy of the three algorithms. In the next section this is being highlighted even more, by generalizing the problems that the algorithms solve and providing a generalized solution.

4 A general algorithm

In this section we assume that the input for all three algorithms is a multigraph G. For Warshall's algorithm one gets the graph representation of the relation as an input, for Kleenes algorithm the transitions and ϵ-moves are being treated as edges and the states as nodes.

4.1 Matrix creation

Not only can the core piece of the algorithm be generalized, but also the step of the matrix creation. Let $K = (M, +, \cdot, 0, 1, {}^*)$ be a Kleene algebra and G be a graph with nodes $N = \{n_0, n_1, \ldots\}$ and edges $E = \{e_0, e_1, \ldots\}$. Furthermore let $w_e(e_i) \in M$ be the weight (i.e. an element of the algebra), $s(e_i)$ the source node and $t(e_i)$ the target node of an edge e_i. Then one can generalize the matrix construction as follows:

$$A_{[i,j]} = \begin{cases} a + w(e_u) + \ldots + w(e_v), & \text{if } i = j \text{, with } e_u, \ldots, e_v \text{ being} \\ & \text{all the edges from } n_i \text{ to } n_j \\ b + w(e_u) + \ldots + w(e_v), & \text{otherwise, with } e_u, \ldots, e_v \text{ being} \\ & \text{all the edges from } n_i \text{ to } n_j \end{cases}$$

For all three algorithms b is the element 0 $(0, \infty, [\varnothing])$ with the intuition that the node is not reachable. The element a is the element 1 for Floyd's and Kleene's algorithm $(0, [\epsilon])$, but Warshall's algorithm differs in this case. Intuitively one would also assume the element 1. But the element 0 has to be used, since 1 would result in computing the *reflexive* transitive closure as the matrix construction would add all reflexive pairs.

By the nature of the generalization and the comparison of the three algorithms, using the algorithm for the *reflexive* transitive closure would probably be more suitable than Warshall's algorithm. Alternatively, one could also introduce another Kleene algebra using three elements. The added element then catches all cases of reflexive pairs that do not have to be added due to transitivity.

But since the algorithm should be used for teaching we stick with the original elements and algebra. Therefore we leave the element for a as an input for the matrix creation algorithm. The following code implements this concept:

Algorithm 4: Matrix creation

```
$n \leftarrow |N|$
$A \leftarrow$ New array of size $n \times n$
for $i \leftarrow 0 \dots (n-1)$ do
    for $j \leftarrow 0 \dots (n-1)$ do
        if $i = j$ then
            $A[i][j] \leftarrow a$ // The element a of the input
        else
            $A[i][j] \leftarrow 0$ // The neutral element of $+$
for edge/transition $e$ in $E$ do
    $A[s(e)][t(e)] \leftarrow A[s(e)][t(e)] + w(e)$ // $+$ operation of $K$
```

4.2 Algorithm

In this section we introduce an algorithm that solves the given problem for Kleene algebras in general. One can then call this algorithm with any Kleene algebra, in our case with the algebras defined in section 2.3, and obtain the desired solution. Since we generalize the algorithm, a generalized definition of the problem that the algorithm actually solves has to be given as well:

Let $K = (M, +, \cdot, 0, 1, *)$ be a Kleene algebra and G be a graph with nodes $N = \{n_0, n_1, \dots\}$ and edges $E = \{e_0, e_1, \dots\}$. Let $w(e_i) \in M$ be the weight (i.e. an element of the algebra) of the edge e_i. Furthermore we call $w_p(p_i)$ the weight of a path $p_i = (e_a, e_b, \dots)$, defined as $w_p(p_i) = w_e(e_a) \cdot w_e(e_b) \cdot \dots$. The $+$ operation is applied on the weights of two paths that are being compared, the $\cdot$ operation will be applied when two paths are being combined one after the other and the $*$ operation will be used to compare all the possible iterations of a path. Then for every pair (n_i, n_j) of nodes the algorithm computes one element $A_{[i,j]} \in M$ satisfying the following condition:

Let Σ be the set that holds a unique symbol for each unique value in $w(e_0), w(e_1), \dots$ and $s : \{w(e_0), w(e_1), \dots\} \rightarrow \Sigma$ the function that maps an edge weight to its symbol and s^{-1} be it's inverse. Now we construct an automaton G' as follows: G' has a state for each node and a transition for each edge of G, where a transition holds the corresponding symbol of the edges weight. We define the function F, mapping regular expressions to elements of the given algebra as follows:

$$F(x) = \begin{cases} 0^K, & \text{if } x = \varnothing \\ 1^K, & \text{if } x = \epsilon \\ s^{-1}(x), & \text{if } x \in \Sigma \\ F(a) +^K F(b), & \text{if } x = a +^R b \\ F(a) \cdot^K F(b), & \text{if } x = a \cdot^R b \\ F(a)*^K, & \text{if } x = a*^R \end{cases}$$

We define r to be the regular expression that defines the language accepted by the automaton G' going from node n_i to node n_j (computed by e.g. Kleene's algorithm). Then $A_{[i,j]} = F(r)$ i.e. the regular expression computed by e.g. Kleene's algorithm for the pair of states (n_i, n_j) mapped to its corresponding algebraic value.

This statement bases on the following perception: We know that Kleene's algorithm computes a regular expression for each pair (s_i, s_j) of states, a generator for all possible words/paths going from s_i to s_j. Kozen proved [10], that there exists an isomorphism between regular expressions and the free Kleene algebra on free generators Σ. Hence we can use regular expressions to argue about all Kleene algebras, without any loss of generality. The general algorithm only differs from Kleene's algorithm in one point: It uses the operations of Kleene algebras instead of the operations of regular expressions. So one could say that the algorithm basically computes a regular expression (equivalent to Kleene's algorithm) and then maps this regular expression to an element of the Kleene algebra using the isomorphism. For a concrete Kleene algebra, this results in evaluating the regular expression using the function F. This enables us to argue about what the produced result expresses in the environment of a given Kleene algebra.

Algorithm 5: General Algorithm

```
1 $n \leftarrow |N|$
2 for $k \leftarrow 0 \dots (n-1)$ do
3     $B \leftarrow A$
4     for $i \leftarrow 0 \dots (n-1)$ do
5         for $j \leftarrow 0 \dots (n-1)$ do
6             $B_{[i,j]} \leftarrow A_{[i,j]} + A_{[i,k]}(A_{[k,k]})^* A_{[k,j]}$ // Operations of $K$
7     $A \leftarrow B$
```

4.3 Proof

In this section we prove that after the general algorithm completed, $A_{[i,j]} = F(r)$ where r denotes the corresponding regular expression for the pair (n_i, n_j) computed as suggested in the previous section 4.2.

First we present an example to motivate the proof. Take the graph G in Figure 3.9. Let G' be the corresponding automaton where the transitions are labeled with the values of the edges. For the pair of nodes (n_0, n_2) Kleene's algorithm returns the regular expression $2 \cdot 3* \cdot 2$. Mapping this expression to the Kleene algebra for Floyd's algorithm using F we get: $F(2 \cdot 3* \cdot 2) = 2 + 0 + 7 = 9$. In the next section we will show, that this results is equivalent to the result computed by Floyd's algorithm itself.

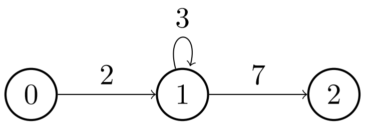


Figure 3.9

Note that this mapping could be performed at any point in the algorithm. One can think of the regular expressions as placeholders, which can at any time be replaced with its corresponding values. So we could compute a part of the algorithm using regular expressions, map them to another Kleene algebra and finish the algorithm with the operations of that algebra.

Now we provide the corresponding proof. We write $A_{k[i,j]}$ for the element $A_{[i,j]}$ after the k-th iteration of the algorithm. Equivalently we write $B_{k[i,j]}$ for the corresponding value computed by Kleene's algorithm for the constructed automaton. It's sufficient to prove that the invariant $A_{k[i,j]} = F(B_{k[i,j]})$ holds before and after each of the k iterations. We prove this by performing induction on k.

Base case $k = 0$:

Let n_i and n_j be two arbitrary nodes of G. By the construction of B it is clear that $B_{0[i,j]} = s(A_{0[i,j]})$. It follows that $F(B_{0[i,j]}) = s^{-1}(B_{0[i,j]}) = A_{0[i,j]}$ since s is a bijection by its construction and $B_{0[i,j]} \in \Sigma$.

Step case $k - 1 \to k$:

Let n_i and n_j be two arbitrary nodes of G. Then

$$B_{k[i,j]} = B_{k-1[i,j]} +^R B_{k-1[i,k]} \cdot^R (B_{k-1[k,k]})*^R \cdot^R B_{k-1[k,j]}$$

Evaluating $F(B_{k[i,j]})$ by using the definition of F and the I.H. we get:

$$\begin{aligned} F(B_{k[i,j]}) &= F(B_{k-1[i,j]} +^R B_{k-1[i,k]} \cdot^R (B_{k-1[k,k]})^{*R} \cdot^R B_{k-1[k,j]}) \\ &= F(B_{k-1[i,j]}) +^K F(B_{k-1[i,k]}) \cdot^K (F(B_{k-1[k,k]}))^{*K} \cdot^K F(B_{k-1[k,j]}) \\ &\overset{\text{I.H.}}{=} A_{k-1[i,j]} +^K A_{k-1[i,k]} \cdot^K (A_{k-1[k,k]})^{*K} \cdot^K A_{k-1[k,j]} \end{aligned}$$

Since $F(B_{k[i,j]})$ is equivalent to the expression that is assigned to $A_{k[i,j]}$ we are done. $\square$

We state another property for regular expression and their corresponding languages, which helps us to prove corollaries in the next section.

Lemma 1: Let x be a regular expression and $w = (w_0, \ldots, w_n)$ be some word. Then $w \in L(x) \iff (w_0 \cdot \ldots \cdot w_n) + x \equiv x$.

Proof: Clearly $L(w_0 \cdot \ldots \cdot w_n) = \{w\}$. Hence $(w_0 \cdot \ldots \cdot w_n) + x \equiv x \iff L((w_0 \cdot \ldots \cdot w_n) + x) = L(x) \iff L(w_0 \cdot \ldots \cdot w_n) \cup L(x) = L(x) \iff \{w\} \cup L(x) = L(x) \iff w \in L(x)$. $\square$

We can now apply F and hence state that $w \in L(x) \iff F(w) + F(x) = F(x) \iff F(w) \leq F(x)$ (partial order of Kleene algebra). Note that for two elements of the Floyd algebra $x \leq_K y$ (order of Kleene algebra) $\iff x \geq_{\mathbb{R}} y$ (order of real numbers) since then $min(x, y) = y$.

4.4 Corollaries

We already proved that the algorithm basically computes a regular expression and evaluates it using the given operations (i.e. the operations of the Kleene algebra the algorithm was called with). It remains to show, that evaluating this regular expression / algebraic term returns the same result as the corresponding algorithm. E.g. evaluating the expression using the Kleene algebra for Floyd's algorithm results in the same matrix as Floyd's algorithm would have computed by itself.

Corollary 1: The general algorithm called with the Kleene algebra for Warshall's algorithm 2.3 is equivalent to Warshall's algorithm.

Proof: Take an arbitrary entry $A_{[i,j]}$ of the result computed by the general algorithm and it's corresponding regular expression x for which $F(x) = A_{[i,j]}$ holds. If n_j is not reachable starting from n_i in G, $x = \varnothing$ must hold by the construction of G'. Hence $F(x) = 0$. If n_j is reachable starting from n_i by the path p, then there must be a corresponding word $w = (w_0, \ldots, w_n)$ in $L(x)$. Trivially $s^{-1}(w_0) = \ldots = s^{-1}(w_n) = 1$ since every edge in the graph has value 1. Hence evaluating with the operations of the Kleene algebra for Warshall's algorithm we get: $F(x) = F((w_0 \cdot^R \ldots \cdot^R w_n) + x) =$

$(s^{-1}(w_0) \wedge \ldots \wedge s^{-1}(w_n)) \vee F(x) = 1 \vee F(x) = 1$. Since the algorithm computes 1 whenever a node is reachable and 0 whenever it isn't, it clearly computes the same result as Warshall's algorithm. Another way to prove this is to insert the concrete operations of the Kleene algebra for Warshall's algorithm in *line 6* of the algorithm: $A_{[i,j]} +^K A_{[i,k]} \cdot^K (A_{[k,k]})*^K \cdot^K A_{[k,j]} = A_{[i,j]} \vee (A_{[i,k]} \wedge A_{[k,j]})$. This trivially computes the same result as the **if** $A_{[i,k]} = 1$ **and** $A_{[k,j]}$ **then** $B_{[i,j]} \leftarrow 1$ **else** $B_{[i,j]} \leftarrow A_{[i,j]}$ of Warshall's algorithm. □

Corollary 2: The general algorithm called with the Kleene algebra for Floyd's algorithm 2.3 is equivalent to Floyd's algorithm.

Proof: Take an arbitrary entry $A_{[i,j]}$ of the result computed by the general algorithm and it's corresponding regular expression x for which $F(x) = A_{[i,j]}$ holds. If there is no path from n_i to n_j in G, $x = \varnothing$ must hold by the construction of G'. Hence $F(x) = \infty$. Otherwise there are some paths from n_i to n_j in G', and there is a corresponding word $w = (w_0, \ldots, w_n)$ in G' for the shortest of these paths (and also for all the others). Let's take an arbitrary word $v \in L(x)$. As $F(v)$ denotes the length of the path that the word v represents it can only be greater than or equal to $F(w)$, the length of the shortest path. Since there must be one word u where $F(u) = F(x)$, trivially $u = w$ must hold. Stated differently: $F(x) = F((w_0 \cdot^R \ldots \cdot^R w_n) + x) = min(s^{-1}(w_0) + \ldots + s^{-1}(w_n), F(x)) = s^{-1}(w_0) + \ldots + s^{-1}(w_n)$ which is the length of the shortest path. Since the algorithm computes ∞ if there is no path and the length of the shortest path otherwise, it computes the same result as Floyd's algorithm. We can also show this by inserting the corresponding operations in *line 6* of the algorithm: $A_{[i,j]} +^K A_{[i,k]} \cdot^K (A_{[k,k]})*^K \cdot^K A_{[k,j]} = min(A_{[i,j]}, (A_{[i,k]} + A[k,j]))$ which is equivalent to the operation performed by Floyd's algorithm. □

Corollary 3: The general algorithm called with the Kleene algebra for Kleene's algorithm 2.3 is equivalent to Kleene's algorithm.

Proof: Take an arbitrary entry $A_{[i,j]}$ of the result computed by the general algorithm and it's corresponding regular expression x for which $F(x) = A_{[i,j]}$ holds. Trivially F reverts the renaming introduced by the construction of G'. It follows directly that $F(x)$ i.e. replacing the unique renamed symbols by their original ones in the computed regular expression returns the regular expression that Kleene's algorithm would compute for the nodes n_i and n_j in G. □

4.5 Complexity

Let $n = |N|$ and $e = |E|$. The matrix construction trivially runs in $\mathcal{O}(n^2 + e)$. The algorithm itself runs in $\mathcal{O}(n^3)$. When seeing both procedures as one algorithm, this adds up to $\mathcal{O}(n^3 + e)$. Note that this runtime is still the same as presented in section 1.4, the only difference is that here we also take the matrix construction into consideration. In most cases $e \leq n^3$ will hold (i.e. for Warshall's algorithm it always does), but since multigraphs and multiple transitions from one state to another are allowed, it's been added here for completeness. For the space complexity we have $\Omega(n^2)$. That's not worse compared to the space complexity of the three single algorithms, it's just stated in a more general way. When running the algorithm with Kleene algebras for Floyd's and Warshall's algorithm, the space complexity remains $\mathcal{O}(n^2)$.

4.6 Simplification

As mentioned in section 1.4 and section 4.5 the length of regular expressions grows exponentially. Hence we aim to replace regular expressions with shorter ones of the same equivalence class, i.e. we try to find a shorter regular expression defining the same language. When running the algorithm with the automaton in Figure 3.5 the entry $A_{[0,1]}$ containing the final solution would hold the following regular expression:

$$\begin{gathered}\varnothing + b + (\epsilon + a)(\epsilon + a)^*(\varnothing + b) + (\varnothing + b + (\epsilon + a)(\epsilon + a)^*(\varnothing + b)) \\ (\epsilon + b + (\varnothing + a)(\epsilon + a)^*(\varnothing + b))^*(\epsilon + b + (\varnothing + a)(\epsilon + a)^*(\varnothing + b))\end{gathered}$$

This regular expression can be simplified to just $a^*b(a^*b)^*$. One can imagine how this can lead to several problems when using an automaton with more than two states and four transitions.

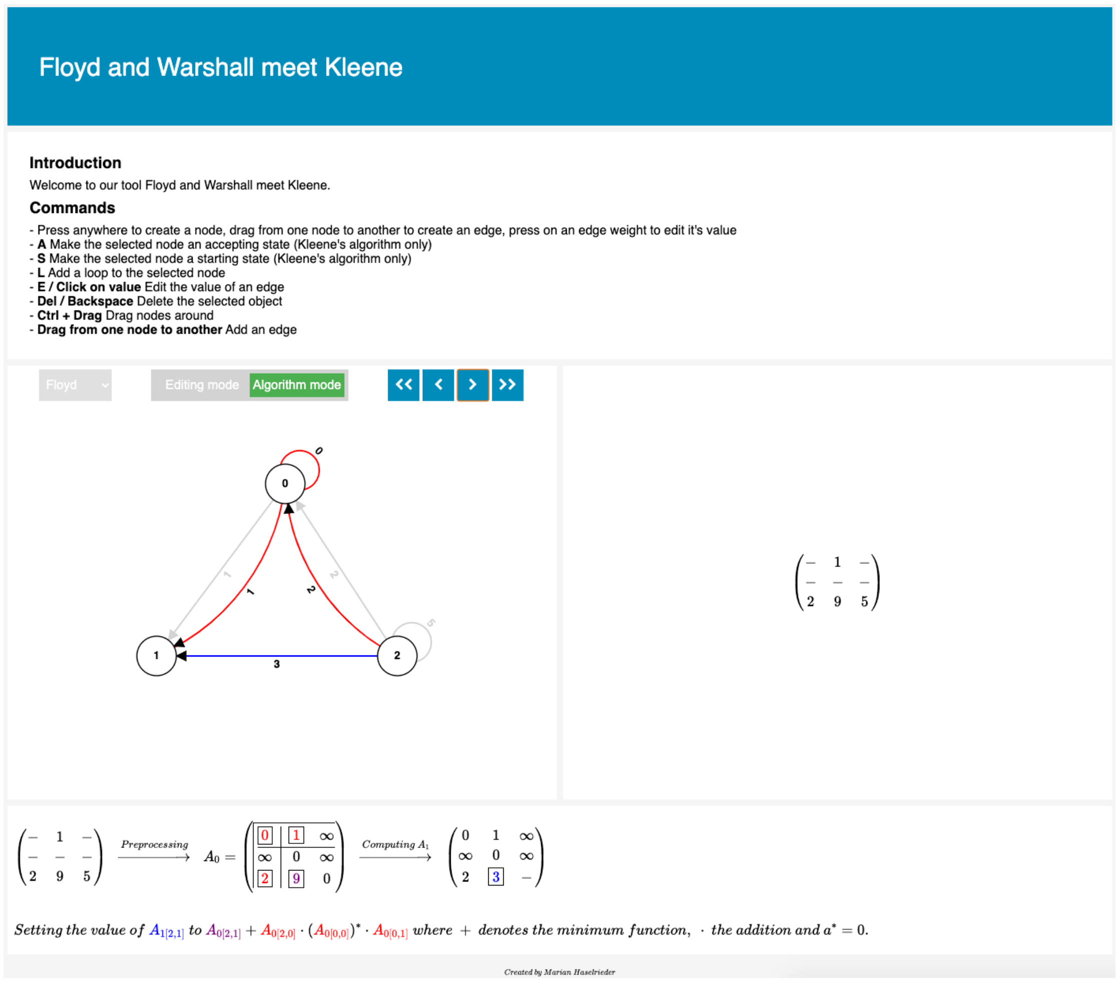
Using the algorithm proposed in section 4.2, this exponential growth mainly happens when using the algebra for regular expressions. For both Floyd's and Warshall's algorithm we have the special case that $x^* = 1$ for all x and hence the expressions do not grow in the same way. For Warshall we always stay in $\mathcal{O}(n^2)$, since we only need one bit for each element. For Floyd's algorithm the longest (not ∞) possible path can't be longer than the sum of all edge weights of the graph which are not ∞. So we can assume that no element takes up more space than this one, hence we remain in $\mathcal{O}(n^2)$. Since one might use the algorithm with other Kleene algebras that have similar growing elements, one might modify the algorithm as follows: Firstly the algorithm only computes algebraic expressions using variables, $+, \cdot$ and * and only after all the final expressions have been produced, they are being evaluated with the respective elements and operations. With this approach, the simplification of the terms can be done directly and generally for all Kleene algebras. The downside is that algorithms, which in most cases have elements of constant size (as Floyd's and Warshall's algorithm), would still take up space in $\mathcal{O}(n^2)$ (worst case exponentially if one cannot simplify properly). Additionally the runtime would increase, since optimal simplification cannot be done in constant time. In any case, the simplification of Kleene algebras and regular expressions is equivalent. Hence only regular expressions are being simplified as part of this thesis and the algorithm proposed in 4.2 is being used.

The most intuitive approach would be a case analysis on the given Kleene algebra and checks whether it matches certain patterns. The axioms 2.1 and the equivalences 2.2 of Kleene algebras immediately provide some easy to implement and fast simplifications.

The removal of the neutral elements whenever possible using (1) and (4) or evaluating to 0 whenever the anihilation by 0 using (6) is possible would be such simplification. In addition to the axioms (1), (4) and (6) 2.1 all the equivalences of section 2.2 have been implemented into our tool.

This might seem like a valid approach, but it fastly reaches it's limits. Already implementing the idempotency of +, namely the simplification $a + a = a$, proves to be quite complex. One cannot fastly decide whether two kleene algebras are equivalent, since they might be of a different structure e.g. $a + (b + c)$ and $c + (b + a)$. More information on this topic can be found in the paper *Testing the Equivalence of Regular Languages* [1] Another problem is that a case analysis expands rapidly, since for each case commutativity, assiociativity etc. have to be added manually. An example would be $(\epsilon+a)* = (a+\epsilon)* = a*$. There are not that many simplifications that can be performed, without checking conditions like equality = or the partial order >=. Just missing a few cases again results in huge equations, all the simplifications just shift everything one or two iterations to the back. Nevertheless this approach is being used in our application since a more efficient approach would be out of the scope of this project.

4 — Web application



Floyd and Warshall meet Kleene

Introduction

Welcome to our tool Floyd and Warshall meet Kleene.

Commands

- Press anywhere to create a node, drag from one node to another to create an edge, press on an edge weight to edit it's value
- **A** Make the selected node an accepting state (Kleene's algorithm only)
- **S** Make the selected node a starting state (Kleene's algorithm only)
- **L** Add a loop to the selected node
- **E / Click on value** Edit the value of an edge
- **Del / Backspace** Delete the selected object
- **Ctrl + Drag** Drag nodes around
- **Drag from one node to another** Add an edge

$$\begin{pmatrix} - & 1 & - \\ - & - & - \\ 2 & 9 & 5 \end{pmatrix}$$

$$\begin{pmatrix} - & 1 & - \\ - & - & - \\ 2 & 9 & 5 \end{pmatrix} \xrightarrow{\textit{Preprocessing}} A_0 = \left(\begin{array}{|c|cc} \boxed{0} & \boxed{1} & \infty \\ \hline \infty & 0 & \infty \\ \boxed{2} & \boxed{9} & 0 \end{array}\right) \xrightarrow{\textit{Computing } A_1} \begin{pmatrix} 0 & 1 & \infty \\ \infty & 0 & \infty \\ 2 & \boxed{3} & - \end{pmatrix}$$

Setting the value of $A_{1[2,1]}$ *to* $A_{0[2,1]} + A_{0[2,0]} \cdot (A_{0[0,0]})^* \cdot A_{0[0,1]}$ *where* $+$ *denotes the minimum function,* $\cdot$ *the addition and* $a^* = 0$.

Created by Marian Haselrieder

1 Functionality

The web application presented here has been developed as a part of this bachelor thesis project. On the one hand it shows that all the algorithms can be replaced with the general algorithm proposed in section 4.2. On the other hand it visualizes step by step how the algorithm works on a given input. Therefore the application can be used while teaching in several different courses e.g. algorithms and datastructures, discrete mathematics, formal languages, automata theory, ...

Graph and automaton construction:

The graph and automaton creation is quite intuitive. One can select between the different algorithms which changes the construction: For Warshall's algorithm edge weights are being disabled, for Floyd's algorithm the allowed values for edges differ from the ones for Kleene's algorithm. Additionally when Kleene's algorithm has been selected, one can also set a node to be a starting or an accepting state. Edges and nodes can obviously be added, dragged, selected and deleted. While constructing the graph, a matrix representation of it will be generated automatically. It stores all values that have been assigned to the different edges. Instead of drawing multiple edges from a node n_i to another node n_j, only one edge which can hold multiple values is being drawn. The figures 4.1 and 4.2 demonstrate this process.

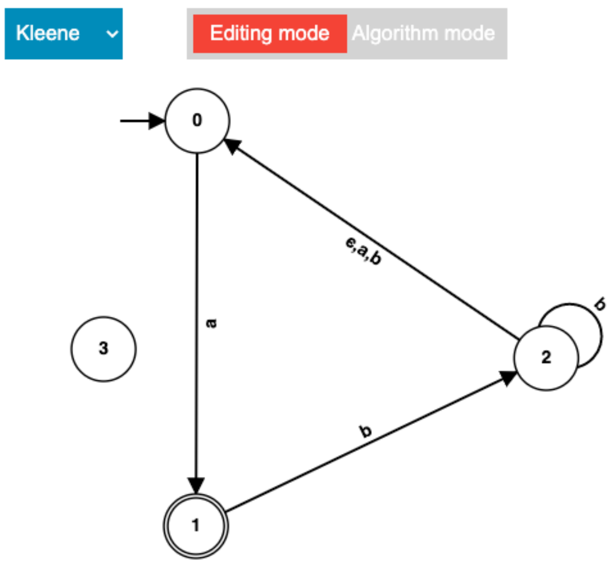


Figure 4.1: Graph construction

$$\begin{pmatrix} - & a & - & - \\ - & - & b & - \\ \epsilon, a, b & - & b & - \\ - & - & - & - \end{pmatrix}$$

Figure 4.2: Construction matrix

Step by step solution:

After constructing a graph, one can switch from the *editing mode* to the *algorithm mode* using the respective toggle. There are multiple ways to step through the algorithm. One can perform a big step, which completes the computation of the current matrix A_k by completing the inner two loops. For the case that all inner iterations have already been stepped through, the next matrix A_{k+1} is being computed. Smaller steps are also possible, which correspond to computing the next value of the current matrix (the value for the next pair of nodes). Of course small and big steps can be performed in both directions.

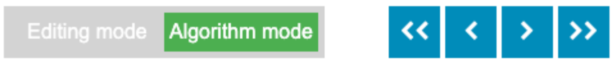


Figure 4.3: Algorithm mode controls

When stepping through the iterations the progress of the algorithm is being visualized in two different places: In the graph and in a step by step matrix representation.

All the edges of the graph become less visible and four additional edges will be drawn. Let's assume we are in the iteration k, i, j. Then the following additional edges will be drawn:

- A red edge from n_i to n_k labeled with the value $A_{k-1[i,k]}$.
- A red loop at the node n_k labeled with the value $A_{k-1[k,k]}$.
- A red edge from n_i to n_k labeled with the value $A_{k-1[k,j]}$.
- A purple edge from n_i to n_j labeled with the value $A_{\boldsymbol{k-1}[i,j]}$.
- A blue edge from n_i to n_j labeled with the value $A_{\boldsymbol{k}[i,j]}$.

This way one can clearly see how the new element $A_{k[i,j]}$ is being formed. It's being formed by introducing an alternative (+ operation) between a newly computed value going through node n_k and the old value. The new value is being computed by sequencing ($\cdot$ operation) the elements of the three red edges, where the * operation is additionally being called on the element $A_{k-1[k,k]}$.

The step by step solution with matrices shows the initial input matrix created during construction, the matrix that has been constructed by the preprocessing or matrix creation steps and all the matrices $A_0, A_1, \ldots$ that have been computed up to now. For the iteration k, i, j the same elements as mentioned above are being highlighted in the matrices A_k and A_{k-1}.

in Figure 4.4 and 4.5 the element ∞ of $A_{1[0,4]}$ is being replaced by the element ∞+8(0)*1 which evaluates to 9 using the Kleene algebra proposed in section 2.3.

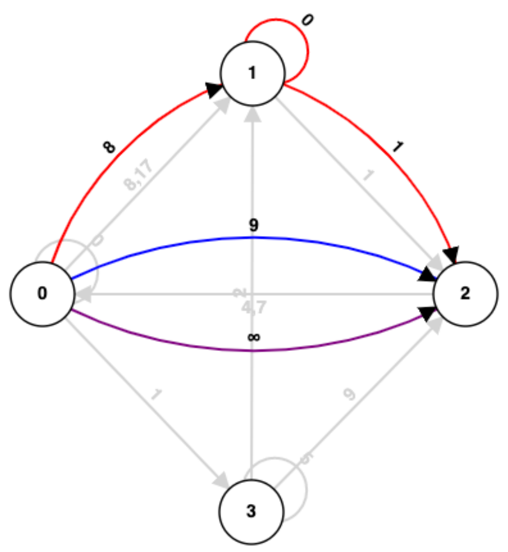


Figure 4.4: Graph step by step solution

$$\begin{pmatrix} 0 & 8,17 & - & 1 \\ - & - & 1 & - \\ 4,7 & - & - & - \\ - & 2 & 9 & 5 \end{pmatrix} \xrightarrow{\textit{Preprocessing}} A_0 = \left(\begin{array}{|c|ccc} \hline 0 & 8 & \infty & 1 \\ \hline \infty & 0 & 1 & \infty \\ 4 & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{array}\right) \rightarrow A_1 = \left(\begin{array}{c|c|cc} 0 & \boxed{8} & \boxed{\infty} & 1 \\ \hline \infty & \boxed{0} & \boxed{1} & \infty \\ \hline 4 & 12 & 0 & 5 \\ \infty & 2 & 9 & 0 \end{array}\right) \xrightarrow{\textit{Computing } A_2} \begin{pmatrix} 0 & 8 & \boxed{9} & - \\ - & - & - & - \\ - & - & - & - \\ - & - & - & - \end{pmatrix}$$

Figure 4.5: Matrix step by step solution

2 Implementation

In this section we firstly give a general overview of the programming languages and libraries that have been used to create the tool. Secondly we give insights into the concrete implementation of the general algorithm. As the core piece of the tool is presented as part of this thesis (with both pseudocode and an implementation) we hardly documented the code of the tool.

2.1 Used technologies

The tool created for this thesis is a webtool implemented using HTML, CSS and Javascript. We decided to build a web-based application, to make the tool portable and easily accessible for everyone. This also implied the usage of HTML, CSS and Javascript. In order to make the tool stable and resistant to any third party code changes, the third party libraries have been directly imported into the project. The following Javascript libraries have been integrated to provide additional functionality:

- **D3.js version 5.16.0 [2]:** "D3.js is a JavaScript library for manipulating documents based on data. D3 helps you bring data to life using HTML, SVG, and CSS." [2] D3.js has been used to create the main *svg* of the application. It's responsible for rendering all graphs and automatas, processing the user's input, the graph creation and partly the visualisation of the step by step solution. Allthough there are multiple libraries providing similar functionality, we chose D3.js because of its comprehensive documentation and intuitive usage.

- **MathJax version 3 [3]:** "Beautiful and accessible math in all browsers: A JavaScript display engine for mathematics that works in all browsers." [3]. The web application uses MathJax to render all mathemathical expressions. It's mainly been used to draw all the matrices of the application, especially the ones of the step by step solution. As we wanted the representation of the matrices to be both dynamic and similar to the Latex design, MathJax was the only but still perfectly fitting option.
- **FontAwesome version 4.7.0 [8]:** FontAwesome provides free to use icons, which have been used to improve the user interface.

The implementation basically works as follows: D3.js is being used to create the input data, then the data is being passed on to the algorithm in combination with the picked Kleene algebra. The algorithm computes the resulting two dimensional arrays (it's a slight variation of the algorithm where one can specify how many iterations should be computed). These results are being passed to the graph in order to progress of the algorithm. Additionally they're being passed to a converter which creates MathJax expressions from the given arrays. These expressions are converted to the matrices which again show the process of the algorithm. Here we only give insight into the algorithmic part of the implementation, since it is the most relevant one.

2.2 The algorithm

Since the tool is a web application, we've created an object oriented implementation of the algorithm using Javascript. As other programming languages provide a cleaner and simpler syntax for object oriented programming, code samples providing the same functionality using the programming language Dart are being used here.

All Kleene algebras have to extend the same abstract class, which demands all operations as well as the identity elements to be defined:

```
abstract class KleeneAlgebra<T> {
  T zero();
  T one();
  T op1(T a, T b);
  T op2(T a, T b);
  T star(T a);
}
```

Note that each algebra demands a certain type, which refers to the type of its elements. In our implementation we use integers for Floyd's algorithm and booleans for Warshall's algorithm. For Kleene's algorithm a manually defined class for regular expressions is being used. The following example shows the algebra for Floyd's algorithm, where *null* is internally being used to represent ∞:

```
class FloydAlgebra extends KleeneAlgebra<int> {

  int zero() => null;

  int one() => 0;

  int op1(int a, int b) {
    if (b == null) return a;
    if (a == null) return b;
    return math.min(a, b);
  }

  int op2(int a, int b) {
    if (a == null || b == null) return null;
    return a + b;
  }

  int star(int a) => 0;
}
```

The algorithm itself takes a Kleene algebra and a two dimensional array containing elements of the algebra as an input. It then computes the solution as described in section 4.2.

```
void algorithm(List<List> inputMatrix, KleeneAlgebra alg) {
  int n = inputMatrix.length;
  List<List> A = clone(inputMatrix);

  for (int k = 0; k < n; k++) {
    List<List> B = clone(A);
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        B[i][j] = alg.op1(
            A[i][j],
            alg.op2(
              alg.op2(
                A[i][k],
                alg.star(A[k][k])),
              A[k][j]
            ));
      }
    }
    A = B;
  }
}
```

Note that no simplification on the algorithmic side is being done here, since everything is directly being evaluated. Nevertheless simplification is definitely necessary for certain Kleene algebras, as e.g. the algebra for regular expressions to avoid algebraic terms taking up exponential space. With this implementation the simplification is directly shifted to the implementation of the operation. See section 4.6 for more details.

5 — Summary, conclusion and outlook

1 Summary

It has been shown that all three algorithms follow the exact same pattern. They generate all possible paths between every pair of nodes of a graph, evaluate a path by the sequence of its edges and afterwards choose between all the evaluations. A Kleene algebra $(A, +, \cdot, 0, 1, *)$ has been provided for each of the algorithms in section 2.3, where the $+$ function refers to the "choice" between values of the algebra, the $\cdot$ function corresponds to "sequencing" elements of the algebra and the $*$ operation refers to the "iteration" of an element. A general algorithm then for each pair of nodes (n_i, n_j) computes one element $A_{[i,j]}$ of the same structure as a regular expression which can intuitively be interpreted as follows: For every path from n_i to n_j it produces an element by applying the $\cdot$ operation on the weights of its edges, then it combines all this elements by using the $+$ operation. Evaluating this term with the given interpretation for $+$, $\cdot$ and $*$ we get the same result with the general algorithm as with the corresponding algorithm for this problem.

2 Conclusion and outlook

Not only is it useful to have one algorithm computing the solution for several different problems, but it also gives insights into their nature and their similarities. Seeing all three algorithms as variations of one and the same, simplifies understanding them and can hence be used when teaching them in all the different courses. Additionally generalizing the algorithm and the concepts opens up the door for finding, reinterpreting and adding similar problems to this collection. However there are also drawbacks when using such an approach: Possible optimizations, space complexity and simple modifications to the algorithms might be affected and complicated by the fact of a generalization. There are still many things that can be improved and many open questions, we only list a few here:

- Is it possible to introduce a simplification system that does not increase the time complexity by much?

- Can one use a different Kleene algebra for Floyd's algorithm, such that it keeps track of the nodes that a computed shortest path passed through? is made out of?
- Multiple algorithms or variations can be added to the provided scheme as computing the reflexive transitive closure, longest path, negative cycles detection, ...
- Clearly undirected graphs do also work with the general algorithm if one converts them into an equivalent directed graph. before running the algorithm. Since the results for the pair (n_i, n_j) and the pair (n_j, n_i) are clearly equivalent, it could be an option to adapt the algorithm to this special case to avoid redundant computations.
- How does the algorithm for *State Elimination* fit into this collection? How much effort would it take to integrate it and can it give further insights into the general problem itself?
- Kleene's algorithm as well as our implementation of the general algorithm could be extended, such that all kind of regular expressions are allowed as edge weights, instead of only symbols.

Bibliography

[1] Marco Almeida, Nelma Moreira, and Rogério Reis. Testing the equivalence of regular languages. *Electronic Proceedings in Theoretical Computer Science*, pages 47–57, 07 2009. `doi:10.4204/EPTCS.3.4`.

[2] Mike Bostock. D3.js - data-driven documents, 2020 (accessed July 29, 2020). `https://d3js.org/`.

[3] MathJax Consortium. Mathjax, 2020 (accessed August 11, 2020). `https://www.mathjax.org/`.

[4] Andrew Goldberg and Tomasz Radzik. A heuristic improvement of the Bellman-Ford algorithm. 1993.

[5] H. Gruber and M. Holzer. Finite automata, digraph connectivity, and regular expression size. *ICALP*, 2008.

[6] Hermann Gruber and Stefan Gulan. Simplifying regular expressions a quantitative perspective. 09 2009. `doi:10.1007/978-3-642-13089-2_24`.

[7] Hermann Gruber and Markus Holzer. From finite automata to regular expressions and back–a summary on descriptional complexity. *Electronic Proceedings in Theoretical Computer Science*, 151, 05 2014. `doi:10.4204/EPTCS.151.2`.

[8] Fonticons Inc. Fontawesome, 2020 (accessed August 14, 2020). `https://fontawesome.com/`.

[9] S. C. Kleene. Representation of events in nerve nets and finite automata. *Automata Studies. (AM-34), Volume 34*, 1951.

[10] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. In *[1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 214–225, 1991.

[11] Nelma Moreira, Davide Nabais, and Rogério Reis. State elimination ordering strategies: Some experimental results. *Electronic Proceedings in Theoretical Computer Science*, pages 139–148, 2010. `doi:10.4204/EPTCS.31.16`.

[12] Georg Moser. Ein Skriptum zur Vorlesung - Diskrete Mathematik. 2018. `http://cl-informatik.uibk.ac.at/teaching/ss18/dm/material/studia.pdf`.

[13] Christos Papadimitriou and Martha Sideri. On the Floyd-Warshall algorithm for logic programs. *The Journal of Logic Programming, Volume 41*, pages 129 – 137, 1999. `doi:https://doi.org/10.1016/S0743-1066(99)00013-8`.

[14] L. Pitt and M. K. Warmuth. The minimum consistent dfa problem cannot be approximated within any polynomial. *[1989] Proceedings. Structure in Complexity Theory Fourth Annual Conference*, pages 230–, 1989.