Verification in Isabelle/HOL of Hopcroft's algorithm for minimizing DFAs including runtime analysis

Vincent Trélat

5th July 2023

Contents

1 Introduction **2**
1.1 Setting up the environment 2
1.2 Hopcroft's algorithm 2
1.3 Modern formalisation 4

2 Proof of correctness **5**

3 Time complexity analysis **6**
3.1 Paper proof 6
3.2 Towards a formal proof 14
3.2.1 Abstract level 14

1 Introduction

The goal of this project was to verify Hopcroft's algorithm for minimizing DFAs in Isabelle/HOL and to analyze its worst-case time complexity using the Refinement Framework. Some considerable work had already been carried out by Peter Lammich and Thomas Türk on the proof of correctness ten years ago in an older version of Isabelle [LT12]. The goal was to port this proof to the latest version of Isabelle and to extend it with a time complexity analysis.

1.1 Setting up the environment

In order to be able to comfortably browse the theories and inspect the proofs, it is recommended to use Isabelle[1]/jEdit. Yet, the project is not part of the Archive of Formal Proofs (AFP) and can only be downloaded from the GitHub repository.

Then, building an image of the main framework is strongly recommended, as it will greatly speed up the compilation of the theories, especially for the time complexity analysis. This may take a few minutes, but only needs to be done once:

```
cd Hopcroft_Verif/Hopcroft_Time
isabelle build -d . -l Isabelle_LLVM_Time Hopcroft_Time.thy
```

Isabelle may then be launched with jEdit by running the following command. This will open the main theory file `Hopcroft_Time.thy` in jEdit and should take less than a minute:

```
isabelle jedit -d . -l Isabelle_LLVM_Time Hopcroft_Time.thy
```

1.2 Hopcroft's algorithm

John E. Hopcroft's algorithm for minimizing deterministic finite automata (DFA) was first presented in his original 1971 paper [Hop71] as a formal algorithm. It has been a major breakthrough in the field of automata theory, since it allowed for minimization in linearithmic[2] time in the worst case[3]. Thanks to this, minimization has become almost costless and thus this algorithm has been widely used in the field of formal verification, e.g. in model checking, since it allows to reduce the size of the automaton to be verified.

[1] The project was carried out using Isabelle2022.

[2] $O(n \log n)$

[3] Other minimization algorithms were presented prior to Hopcroft, e.g. by Moore (quadratic) or Brzozowski (exponential).

The formal idea behind the algorithm is to partition the set of states of the DFA into equivalence classes, i.e. sets of states that are equivalent[4] with respect to the language they recognize. The algorithm starts with two partitions, namely the set of final states and the set of non-final states. Then, it iteratively refines the partitions by splitting them into smaller ones until no more refinement is possible. The algorithm is guaranteed to terminate since the number of partitions is finite and strictly decreases at each iteration. The equivalence class of the set of states of the DFA is then the set of partitions, so that any two different sets of states from that partition recognize different languages.

Algorithm 1: Hopcroft's original formal algorithm

Data: a finite DFA $\mathcal{A} = (\mathcal{Q}, \Sigma, \delta, q_0, \mathcal{F})$
Result: the equivalence class of $\mathcal{Q}$ under state equivalence

Construct $\delta^{-1}(q, a) := \{t \in \mathcal{Q} \mid \delta(t, a) = q\}$ for all $q \in \mathcal{Q}$ and $a \in \Sigma$;
Construct $P_1 := \mathcal{F}$, $P_2 := \mathcal{Q} \setminus \mathcal{F}$ and $s_{a,i} := \{q \in P_i \mid \delta^{-1}(q, a) \neq \varnothing\}$ for all $i \in \{1, 2\}$ and $a \in \Sigma$;
Let $k := 3$;
For all $a \in \Sigma$, construct $L_a := \underset{1 \leq i < k}{\arg\min} \ |s_{a,i}|$;
while $\exists a \in \Sigma, L_a \neq \varnothing$ **do**
  Pick $a \in \Sigma$ such that $L_a \neq \varnothing$ and $i \in L_a$;
  $L_a := L_a \setminus \{i\}$;
  forall $j < k, \exists q \in P_j, \delta(q, a) \in s_{a,i}$ **do**
    $P'_j := \{t \in P_j \mid \delta(t, a) \in s_{a,i}\}$ and $P''_j := P_j \setminus P'_j$;
    $P_j := P'_j$ and $P_k := P''_j$; construct $s_{a,j}$ and $s_{a,k}$ for all $a \in \Sigma$ accordingly ;
    For all $a \in \Sigma$, $L_a := \begin{cases} L_a \cup \{j\} & \text{if } j \notin L_a \wedge |s_{a,j}| \leq |s_{a,k}| \\ L_a \cup \{k\} & \text{otherwise} \end{cases}$;
    $k := k + 1$;
  end
end

Algorithm 1 is a direct translation of the original algorithm from [Hop71] with only slight changes in the notations.

[4]Two states are equivalent if they accept the same language.

1.3 Modern formalisation

The algorithm is now usually given in a more mathematical and formalised way[5], which loosens up the choice for an implementation. The algorithm is given below in Algorithm 2.

Algorithm 2: Hopcroft's algorithm in a modern style

```
Data: a finite DFA A = (Q, Σ, δ, q_0, F)
Result: the language partition P_ℓ
if F = ∅ ∨ Q \ F = ∅ then
    return Q
else
    P := {F, Q \ F} ;
    W := {(a, min{F, Q \ F}), a ∈ Σ} ;
    while W ≠ ∅ do
        Pick (a, C) from W and remove it;
        forall B ∈ P do
            Split B with (a, C) into B_0 and B_1 ;
            P := (P \ {B}) ∪ {B_0, B_1} ;
            forall b ∈ Σ do
                if (b, B) ∈ W then
                    W := (W \ {(b, B)}) ∪ {(b, B_0), (b, B_1)} ;
                else
                    W := W ∪ {(b, min{B_0, B_1})} ;
                end
            end
        end
    end
end
```

We justify this transformation. First, instead of storing indices for the blocks, we directly deal with sets and explicitly work with a partition $\mathcal{P}$ of $\mathcal{Q}$. Then, we define splitters.

Definition 1. Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, I, F)$ be a DFA. Let P be a partition of Q. Let $B \in P$. For $a \in \Sigma$ and $C \in P$ we say that (a, C) splits B or is a splitter of B if

$$\exists q_1, q_2 \in B \quad \delta(q_1, a) \in C \wedge \delta(q_2, a) \notin C.$$

[5] see for example [EB23]

Remark 1. If (a, C) is a splitter of B, P can be updated to $P \setminus \{B\} \cup \{B', B''\}$, where

$$B' := \{q \in B \mid \delta(q, a) \in C\} \text{ and } B'' := B \setminus B'.$$

Definition 2. For a splitter $s \in \Sigma \times P$, we define its size $\|s\|$ as the size of its second component, i.e. if $s = (a, C)$, then $\|s\| := |C|$.

In Algorithm 1, picking an index $i \in L_a$ corresponds to picking a set $s_{a,i}$, i.e. – by definition – the subset of P_i of states having at least one predecessor. We cannot directly replace L_a and $s_{a,i}$ by a set of splitters (a, P_i) as is, however the next step in the algorithm is to go over all blocks in the partition that have a successor in $s_{a,i}$. If such a block B is found, it is split into two blocks, namely the set of states having a successor in $s_{a,i}$ and the others. Overall, we look for B such that P_i has a predecessor in B. We add the condition that B must also have at least one successor not in P_i to avoid splitting B into $\{B, \varnothing\}$. Since this is equivalent to splitting B with (a, P_i), we can replace L_a by a set of splitters (a, P_i). Since the symbol a is chosen such that $L_a \neq \varnothing$, we can define a workset $\mathcal{W}$ of all splitters, with all symbols of Σ. Therefore, L_a is empty if and only if there is no a-splitter in $\mathcal{W}$. Because testing emptiness is equivalent to testing membership, and because of the existential quantifier in the definition of splitters, the $s_{a,i}$ are no longer needed.

We can explicitly write the trivial cases where either $\mathcal{F}$ or $\mathcal{Q} \setminus \mathcal{F}$ is empty to improve performance for this specific case. This is not done in Algorithm 1 but this does not produce wrong results. If one of the two sets is empty, one of the $s_{a,1}$ or $s_{a,2}$ will be empty for all $a \in \Sigma$, and the *while* loop will be entered $|\Sigma|$ times but the inner *for* loop will never be entered. This results in a useless $O(|\Sigma|)$ computation. Another difference in that case is that it would return the partition $\min\{\mathcal{F}, \mathcal{Q} \setminus \mathcal{F}\}$ – where min returns the set with the least number of elements – which we avoid in the modern version by just returning $\mathcal{Q}$.

2 Proof of correctness

Since the proof of correctness is not the purpose of this paper, we do not go into details. John E. Hopcroft's original paper [Hop71] contains a proof of correctness for his low-level algorithm, however the ideas behind the invariants are the same for the formalisation.

The proof of correctness for Hopcroft's algorithm for DFA minimization typically involves two main components: partition refinement and equivalence checking.

The basic idea of the proof is to show that the algorithm correctly partitions the states of the input DFA into equivalence classes, where states within the same class are indistinguishable from each other with respect to the language recognized by the DFA. This partition represents the minimal DFA that recognizes the same language as the input DFA.

The algorithm starts with an initial partition that distinguishes accepting states from non-accepting states. Then, it iteratively refines this partition by considering pairs of states that are currently in different equivalence classes and determining if they can be distinguished by a symbol from the input alphabet.

During each iteration, the algorithm examines each symbol in the alphabet and checks if it distinguishes any pair of states in different equivalence classes. If a symbol does distinguish a pair, it splits the equivalence class containing those states into two new classes. This process continues until no more splits can be made, and the resulting partition represents the minimal DFA.

To prove the correctness of Hopcroft's algorithm, one needs to demonstrate two key properties:

- the resulting partition is a valid equivalence relation, meaning it satisfies the properties of reflexivity, symmetry, and transitivity,
- the resulting partition is indeed the minimal partition that correctly distinguishes states based on their language equivalence.

The proof typically involves reasoning about the behavior of the algorithm during each iteration, considering how the partition is refined inductively and ensuring that it converges to the minimal partition.

In the Refinement Framework, the algorithm is first described at the abstract level, so its correctness can be proved independently of any implementation details. Then, the algorithm is progressively refined into a concrete implementation, and the correctness of each refinement until the implementation is proved by showing that it refines the abstract algorithm.

3 Time complexity analysis

3.1 Paper proof

In a first time, we focus on the original algorithm presented in Algorithm 1 in order to work on the arguments given in [Hop71]. The data structures used at that time were mostly linked lists, but let us rather give some requirements

for the data structures instead of actual implementations. The goal is to show that the algorithm can be executed in $O(m \cdot n \log n)$ time, where m is the number of symbols in the alphabet and n is the number of states in the DFA.

The following requirements come directly from [Hop71] and are specific to the algorithm presented in Algorithm 1.

Requirement 1. Sets such as $\delta^{-1}(q, a)$ and L_a must be represented in a way that allows $O(1)$ time for addition and deletion in "front position"[6].

Requirement 2. Vectors must be maintained to indicate whether a state is in a given set.

Requirement 3. Sets such as P_i must be represented in a way that allows $O(1)$ time for addition and deletion at any given position.

Remark 2. From Req. 2 and Req. 3, we can show that for a state q in the representation of a set P_i or $s_{a,i}$, its "position" – in the implemented data structure – must be determined in $O(1)$ time.

Lemma 1. In Algorithm 1, lines 1 to 4 can be executed in $O(|\Sigma| \cdot |\mathcal{Q}|)$ time.

Proof. The non trivial part is the computation of the inverse transition function $\delta^{-1}(q, a)$, for all $q \in \mathcal{Q}$ and $a \in \Sigma$. This can be done in $O(|\Sigma| \cdot |\mathcal{Q}|)$ time by iterating over Σ and traversing the automaton (e.g. with a DFS) while keeping track of the predecessor at each step. ■

Lemma 2. An iteration of the *while* loop in both algorithms for a splitter $s = (a, C)$ takes a time proportional to the number of transitions terminating in C and the number of symbols in the alphabet, i.e. $\Theta\left(|\Sigma| \cdot |\delta^{-1}(C, a)|\right)$ time.

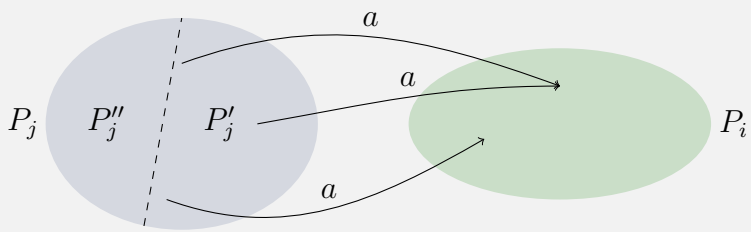
Proof. We start by showing it for Algorithm 1. We pick $a \in \Sigma$ such that $L_a \neq \varnothing$ and an $i \in L_a$. We need to examine all $j < k$ such that $\exists q \in P_j, \delta(q, a) \in s_{a,i}$ to construct the sets corresponding to splitting the block P_j w.r.t. a and P_i.

[6] This does not make sense for sets and is specific for a data structure. What is meant here is a bit informal and designates a position directly accessible, e.g. the head for a stack, the value of the root for a tree, etc.

Let $j < k$. From the definition of $s_{a,i}$, we obtain the following:

$$\begin{aligned}\exists q \in P_j, \delta(q,a) \in s_{a,i} &\iff \exists q \in P_j, \delta(q,a) \in P_i \wedge \underbrace{\delta^{-1}(\delta(q,a),a) \neq \varnothing}_{\text{true}} \\ &\iff \exists q \in P_j, \delta(q,a) \in P_i\end{aligned}$$

Which corresponds to finding states in P_j having an outgoing a-transition to a state in P_i, as represented in the following scheme:



This set of states can be expressed via the inverse transition function:

$$\{q \in P_j \mid \delta(q,a) \in P_i\} = \left(\bigcup_{q' \in P_i} \delta^{-1}(q',a)\right) \cap P_j$$

Since δ^{-1} was already computed in the first step of the algorithm, we can determine using req. 3 whether a state of $\bigcup_{q \in P_i} \delta^{-1}(q,a)$ is also in P_j in $\Theta(1)$ time.

Thus, instead of examining P_j for all $j < k$, we rather go through the table of δ^{-1} and for each state q such that $\delta(q,a) \in P_i$, we know from req. 2 that we can determine the index $j < k$ (because there are k blocks) of the block P_j containing q in $\Theta(1)$ time. The sets P_j' and $P_j'' = P_k$ resulting from the partition can be constructed on the fly without any additional time cost. The construction of the sets $s_{b,j}$ and $s_{b,k}$ as well as the update of L_b for all $b \in \Sigma$ can also be done on the fly but require $\Theta(1)$ time for each symbol $b \in \Sigma$ and thus add up to a total of $\Theta(|\Sigma|)$ time.

VT: the on-the-fly computation part is a little hand-waving...

Overall, one iteration of the loop runs in time

$$\Theta\left(|\Sigma| \cdot \left|\bigcup_{q \in P_i} \delta^{-1}(q,a)\right|\right)$$

We now show the result for Algorithm 2. A splitter $s =: (a, C)$ is picked from $\mathcal{W}$. Then, we iterate over all blocks $B \in \mathcal{P}$ that may be split with s. Let $B \in \mathcal{P}$.

$$\begin{aligned} s \text{ splits } B &\iff \exists q_1, q_2 \in B, \delta(q_1, a) \in C \wedge \delta(q_2, a) \notin C \\ &\iff \left(\bigcup_{q \in C} \delta^{-1}(q, a) \right) \cap B \neq \varnothing \wedge \left(\bigcup_{q \in B} \delta(q, a) \right) \setminus C \neq \varnothing \end{aligned}$$

VT: we may express the right-hand conjunct in terms of δ^{-1} in order to prepare the paragraph that follows (But it wouldn't be the same union because you have to start from states outside C)

VT: The second equation may be a little too quick, especially for the right-hand conjunct.

Likewise, instead of examining all blocks $B \in \mathcal{P}$, we go through δ^{-1} and we update the splitters for all symbols in Σ and one iteration of the loop runs in time

$$\Theta \left(|\Sigma| \cdot \left| \bigcup_{q \in C} \delta^{-1}(q, a) \right| \right)$$

■

For the sake of simplicity, we now denote $\bigcup_{q \in C} \delta^{-1}(q, a)$ by $\overset{\leftrightarrow a}{C}$.

We will now justify the logarithmic factor in the time complexity. We briefly explain why a logarithm stands out and prove the statement by induction. The idea is that for each symbol $a \in \Sigma$, a state $q \in \mathcal{Q}$ can be in at most one of the splitters in $\mathcal{W}$. When the loop iterates over this splitter, it will split the block and keep the smaller one, whose size will be at most the size of the splitter divided by two. This means that a splitter s can be processed at most $\log \|s\|$ times. We now properly state and prove the property by induction.

We see splitters as updatable program variables. This means that $s_{a,i}$ as a set may differ along the loop, however we know there exists some $q \in \mathcal{Q}$ such that $s_{a,i}$ is the unique set containing q throughout the execution. This gives a way to characterize splitters throughout the whole execution.

Lemma 3. Any splitter in the workset $s \in \mathcal{W}$ is processed – i.e. picked at the beginning of the *while* loop – at most $\lfloor \log \|s\| \rfloor$ times.

Proof. We first show the following statement:

During an iteration, the chosen splitter s will either be removed from the set of splitters or its size will be reduced by at least $\frac{\|s\|}{2}$ after the iteration[a].

Let $\{P_1, \cdots, P_\ell\}$ be the current partition and let $s =: (a, C)$ be the picked splitter. Thus, s is no longer in the set of splitters $\mathcal{W}$. Since we go over all splittable blocks, C may also be split by s.

- If C is not split by s, then s was already removed from the splitters. Note that it can be added again later if it is split by another splitter.
- If C is split into C' and C'', then $(a, \min\{C', C''\})$ is added to the splitters and its size is at most $\frac{\|s\|}{2}$.

Therefore, since a splitter cannot be empty, s can be processed at most m times where m is such that

$$1 \leq \frac{\|s\|}{2^m} < 2$$

which is equivalent to

$$m \leq \log \|s\| < m + 1 \quad \text{i.e.} \quad \lfloor \log \|s\| \rfloor = m$$

■

[a] That is a bit informal because, the original splitter is actually removed and another "similar" splitter (cf the paragraph above the statement of the lemma) of size at most $\frac{\|s\|}{2}$ will be added.

Lemma 4. Any block B resulting from a split and not added as a splitter is processed at most $\lfloor \log \frac{|B|}{2} \rfloor$ times.

Proof. Let $P \in \mathcal{P}$ be a block in the partition and $s =: (a, C)$ be a splitter such that s splits P into S and B so that (a, S) is added to the workset and B is not. In order for B to be processed, some (b, B) must be added to the workset.

Since the only way to create a fresh splitter is to split B, this means that there exists some later step in the loop such that some splitter (b, B') splits B into B_1 and B_2 and $(b, \min\{B_1, B_2\})$ is added to the

workset.

From lemma 3, we know that this splitter can be processed at most $\lfloor \log |\min\{B_1, B_2\}| \rfloor$ times. Since $|\min\{B_1, B_2\}| \leq \frac{|B|}{2}$ and since there is no splitter containing B in the workset, we obtain that B can be processed at most $\lfloor \log \frac{|B|}{2} \rfloor$ times.

■

Lemma 5. Let us consider some step in the algorithm such that $\mathcal{P}$ is the current partition. We give an estimation of the total time spent in the loop until termination as an upper bound:

$$T := \theta \cdot \left(\sum_{(a,C) \in \mathcal{W}} |\overset{\hookleftarrow a}{C}| \log |C| + \sum_{(a,B) \in (\Sigma \times \mathcal{P}) \setminus \mathcal{W}} |\overset{\hookleftarrow a}{B}| \log \frac{|B|}{2} \right)$$

where θ is the constant of proportionality that may be obtained from lemma 2.

Proof. We show the result by induction over the steps.

Base case: the current partition is $\{P_1, P_2\}$ and we may assume w.l.o.g. that P_1 is the smaller set, so that $\mathcal{W} = \{(a, P_1), a \in \Sigma\}$. We have to show that the total time spent in the loop is bounded by

$$T = \theta \cdot \sum_{a \in \Sigma} \left(|\overset{\hookleftarrow a}{P_1}| \log |P_1| + |\overset{\hookleftarrow a}{P_2}| \log \frac{|P_2|}{2} \right)$$

We know from lemma 2 that an iteration of the loop for (a, P_i) takes $\theta |\overset{\hookleftarrow a}{P_i}|$ time.

- From lemma 3, for all $a \in \Sigma$, (a, P_1) can be processed at most $\log \|(a, P_1)\| = \log |P_1|$ times[a], hence the total time for (a, P_1) is bounded by $\theta |\overset{\hookleftarrow a}{P_1}| \log |P_1|$ and thus the total time for any splitter (a, P_1) is bounded by:
$$\theta |\overset{\hookleftarrow a}{P_1}| \log |P_1|$$
- From lemma 4, for all $a \in \Sigma$, (a, P_2) can be processed at most $\log \frac{\|(a,P_2)\|}{2} = \log \frac{|P_2|}{2}$ times. Thus, the total time for any non-

splitter (a, P_2) is bounded by:

$$\theta|\overleftrightarrow{P_2}^a|\log\frac{|P_2|}{2}$$

Inductive step: Let $\{P_1, \cdots, P_\ell\} := \mathcal{P}$ be the current partition. The induction hypothesis states that the estimate for the total time spent in the loop until termination for a is bounded by

$$T := \theta\left(\sum_{(a,C)\in\mathcal{W}} |\overleftrightarrow{C}^a|\log|C| + \sum_{(a,B)\in\Sigma\times\mathcal{P}\setminus\mathcal{W}} |\overleftrightarrow{B}^a|\log\frac{|B|}{2}\right)$$

By going through one more step, some blocks of $\mathcal{P}$ may be split and we define a new estimate $\hat{T}$ over this new partition and we have to show that $\hat{T} \leq T$.

Suppose some $B \in \mathcal{P}$ is split into B' and B'' by any splitter. We have to consider the cases where (a, B) is a splitter or not for all $a \in \Sigma$. Let $a \in \Sigma$.

- $(a, B) \in \mathcal{W}$: $\mathcal{W}$ is updated as follows:

  $$\mathcal{W} := \mathcal{W} \setminus \{(a, B)\} \cup \{(a, B'), (a, B'')\}$$

  Thus, instead of taking a time $\theta|\overleftrightarrow{B}^a|\log|B|$ time for (a, B), it now takes a time bounded by:

  $$\theta|\overleftrightarrow{B'}^a|\log|B'| + \theta|\overleftrightarrow{B''}^a|\log|B''|$$

  Finally, we show the following:

  $$|\overleftrightarrow{B'}^a|\log|B'| + |\overleftrightarrow{B''}^a|\log|B''| \leq |\overleftrightarrow{B}^a|\log|B|$$

  From $B = B' \cup B''$ and because the automaton is deterministic,

we get $|B| = |B'| + |B''|$ and $|\overleftrightarrow{B}^a| = |\overleftrightarrow{B'}^a| + |\overleftrightarrow{B''}^a|$, hence:

$$
\begin{aligned}
&|\overleftrightarrow{B'}^a| \log |B'| + |\overleftrightarrow{B''}^a| \log |B''| - |\overleftrightarrow{B}^a| \log |B| \\
&= |\overleftrightarrow{B'}^a| \log |B'| + (|\overleftrightarrow{B}^a| - |\overleftrightarrow{B'}^a|) \log (|B| - |B'|) - |\overleftrightarrow{B}^a| \log |B| \\
&= |\overleftrightarrow{B'}^a| \log \frac{|B'|}{|B| - |B'|} + |\overleftrightarrow{B}^a| \log \frac{|B| - |B'|}{|B|} \\
&\leq |\overleftrightarrow{B'}^a| \log \frac{|B|}{|B| - |B'|} + |\overleftrightarrow{B}^a| \log \frac{|B| - |B'|}{|B|} \\
&= (|\overleftrightarrow{B}^a| - |\overleftrightarrow{B'}^a|) \log \underbrace{\frac{|B| - |B'|}{|B|}}_{\leq 1} \leq 0
\end{aligned}
$$

- $(a, B) \notin \mathcal{W}$: $(a, \min\{B', B''\})$ is added to the set of splitters. We may assume w.l.o.g. that B' is the smaller set. Thus, the corresponding term in the sum $|\overleftrightarrow{B}^a| \log \frac{|B|}{2}$ is updated as follows:

$$
\theta \underbrace{|\overleftrightarrow{B'}^a| \log |B'|}_{\text{lemma 3}} + \theta \underbrace{|\overleftrightarrow{B''}^a| \log \left(\frac{|B''|}{2} \right)}_{\text{lemma 4}}
$$

Since $B = B' \cup B''$, the sum above can be written as:

$$
\theta |\overleftrightarrow{B'}^a| \log |B'| + \theta (|\overleftrightarrow{B}^a| - |\overleftrightarrow{B'}^a|) \log \left(\frac{|B| - |B'|}{2} \right)
$$

By using the fact that $|B'| \leq \dfrac{|B|}{2}$, we have:

$$
\begin{aligned}
&|\overleftrightarrow{B'}^a| \log |B'| + (|\overleftrightarrow{B}^a| - |\overleftrightarrow{B'}^a|) \log \left(\frac{|B| - |B'|}{2} \right) - |\overleftrightarrow{B}^a| \log \frac{|B|}{2} \\
&\leq |\overleftrightarrow{B'}^a| \log \frac{|B|}{2} + (|\overleftrightarrow{B}^a| - |\overleftrightarrow{B'}^a|) \log \left(\frac{|B| - |B'|}{2} \right) - |\overleftrightarrow{B}^a| \log \frac{|B|}{2} \\
&\leq (|\overleftrightarrow{B}^a| - |\overleftrightarrow{B'}^a|) \log \left(\frac{|B| - |B'|}{|B|} \right) \leq 0
\end{aligned}
$$

Overall, splitting any block of $\mathcal{P}$ does not increase the total time spent over the loop. By iterating over all blocks, we obtain that

the total time $\hat{T}$ spent in the loop from this new iteration until termination is such that $\hat{T} \leq T$.

■

[a]We drop the floor operator for simplicity. Note that since we are giving upper bounds, this is completely valid.

Theorem 1. Algorithm 1 and Algorithm 2 run in $O\left(|\Sigma| \cdot |\mathcal{Q}| \log |\mathcal{Q}|\right)$ time.

Proof. From lemma 5, we obtain in particular that for the initial partition the time spent in the loop until termination is bounded by:

$$T = \theta \cdot \sum_{a \in \Sigma} \left(|\overleftrightarrow{P_1}^a| \log |P_1| + |\overleftrightarrow{P_2}^a| \log \frac{|P_2|}{2} \right)$$

Thus using concavity of the logarithm, the following holds:

$$T \leq \theta \sum_{a \in \Sigma} (\underbrace{|\overleftrightarrow{P_1}^a| + |\overleftrightarrow{P_2}^a|}_{\leq 2|\overleftrightarrow{\mathcal{Q}}^a|}) \log(\underbrace{|P_1| + |P_2|}_{=|\mathcal{Q}|}) \leq 2\theta |\Sigma| |\mathcal{Q}| \log |\mathcal{Q}|$$

■

3.2 Towards a formal proof

The objective of the project is to obtain a formal proof for the time complexity of the algorithm using the Refinement Framework with its extension to runtime analysis. We follow a top-down strategy, from the abstract level to the implementation.

3.2.1 Abstract level

The purpose of this section is to explain how the analysis at the abstract level is structured without adding too many details about Isabelle.

Setting up the framework: First, we define the algorithm on the abstract level using the NREST framework as shown below:

```
definition Hopcroft_abstractT where
Hopcroft_abstractT 𝒜 ≡
  if (check_states_empty_spec 𝒜) then mop_partition_empty else
  if (check_final_states_empty_spec 𝒜) then
     mop_partition_singleton (𝒬 𝒜) else
    do {
      PL ← init_spec 𝒜;
      (P, _) ← monadic_WHILEIET (Hopcroft_abstract_invar 𝒜)
          (λ s. estimate_iteration 𝒜 s) check_b_spec
              (Hopcroft_abstract_f 𝒜) PL;
      RETURNT P
    }
```

It is parameterized by abstract costs for each operation, like checking emptiness, returning an empty partition or a singleton, initializing the state[7], etc. For the sake of simplicity and since names are rather self-explanatory, we only give the definition of `check_states_empty_spec` as an example and detail the *while* loop. Checking emptiness for the set of states costs one coin of the currency `check_states_empty` and can be defined as follows:

```
definition check_states_empty_spec 𝒜 ≡
  consume (RETURNT (𝒬 𝒜 = {})) (cost ''check_states_empty'' 1)
```

The purpose of using currencies is to be able to give an abstract description of the costs, and then to instantiate them with concrete values in the implementation which will be expressed in terms of concrete elementary costs like the number of comparisons or assignments. For example, if the set of states is represented as a tree `t`, there should be a method or an attribute `t.isempty` kept up to date during the execution of the algorithm, so that the cost of checking emptiness is equal to the cost of a function call plus the cost for reading memory at the location of the boolean `t.isempty`, that is constant time.

The *while* loop requires to be provided with an invariant, an estimation of the total amount of resource to be consumed – here time, and this estimation has to decrease over an iteration, the loop condition, the function to be executed in the loop body and the initial state. The loop body defined in `Hopcroft_abstract_f` is given below:

[7]States in this setting are tuples (P, L) where P is a partition of the set of states $\mathcal{Q}$ of $\mathcal{A}$ together with a workset L of splitters.

```
definition Hopcroft_abstract_f where
Hopcroft_abstract_f 𝒜 =
  λPL. do {
    ASSERT (Hopcroft_abstract_invar 𝒜 PL);
    (a,p) ← pick_splitter_spec PL;
    PL' ← update_split_spec 𝒜 PL a p;
    RETURNT PL'
  }
```

The main operation, i.e. splitting blocks of the partition and updating the workset accordingly is specified in `update_split_spec`. As written in the informal paper proof of lemma 2, this operation is a bit tricky since both splitting and updating are performed on-the-fly at the same time, thus they are gathered together for the moment. The definition of `update_split_spec` is given below:

```
definition update_split_spec 𝒜 ≡ λ(P,L) a p.
  SPEC (λ(P', L').
    Hopcroft_update_splitters_pred 𝒜 p a P L L' ∧ P' =
        Hopcroft_split 𝒜 p a {} P
  ) (λ_. cost ''update_split'' (enat (|Σ 𝒜| * | preds 𝒜 a p |)))
```

where `preds 𝒜 a p` represents the set of all predecessors $\bigcup_{q\in p} \delta^{-1}(q,a) = \overleftrightarrow{p}^a$ of all states of p labelled by a. The cost specified by this definition stems from the result of lemma 2, namely that the cost for splitting and updating the partition and the workset is $\Theta(|\Sigma| \cdot |\overleftrightarrow{p}^a|)$.

The most important part of this definition is the estimate of the cost for the loop, i.e. the amount of resource consumed by the loop. From lemma 5, we have the following estimate:

$$T = \theta \cdot \left(\sum_{(a,C)\in\mathcal{L}} |\overleftrightarrow{C}^a| \log |C| + \sum_{(a,B)\in(\Sigma\times\mathcal{P})\setminus\mathcal{L}} |\overleftrightarrow{B}^a| \log \frac{|B|}{2} \right) \tag{1}$$

The constant θ comes from lemma 2 and represents constant costs inherent to the loop body, like the cost of checking emptiness, calling a function, etc. In Isabelle, this factor θ is defined following the definition of `Hopcroft_abstract_f` and `check_b_spec` as follows:

```
definition cost_1_iteration ≡
  cost ''call'' 1 + cost ''check_l_empty'' 1 + cost ''if'' 1 +
  cost ''pick_splitter'' 1 + cost ''update_split'' 1
```

Then, given a suitable definition `estimate` for the inner expression in T – i.e. the "raw" estimate – the estimate for the loop is defined as the product[8] of this constant factor and the inner expression, as follows:

```
definition estimate_iteration where
  estimate_iteration 𝒜 PL ≡ cost_1_iteration *m estimate 𝒜 PL
```

Finding a suitable definition for the estimate: Finding a suitable definition for the inner expression in T is not straightforward. The first attempt was to define it as follows:

```
definition estimate1 𝒜 ≡ λ(P,L).
  ∑{(card(preds 𝒜 (fst s) (snd s)))*log(card(snd s)) | s. s ∈ L} +
  ∑{(card(preds 𝒜 (fst s) (snd s)))*log(card(snd s)/2) | s. s ∈ Σ
      𝒜 × P - L}
```

The problem with this definition is that the function $s := (a, C) \mapsto |\overleftarrow{C}^{a}| \log |C|$ has no reason to be injective either on L or its complement and thus the set comprehension expressions within the sums may actually contain duplicates that are merged together. One way to fix this is to sum over multisets. Multisets can also be defined via set comprehension in Isabelle, however having a structure allowing for induction is often more convenient. Thus, we prefer lists to multisets.

A very convenient way to link lists and multisets is to use permutations, which are defined in Isabelle using multisets: a list ℓ is a permutation of a list ℓ' if and only if ℓ and ℓ' have the same multiset, i.e. in Isabelle if and only if `mset` ℓ `=` `mset` ℓ'. We can lift this definition[9] to permutations between lists and sets with the help of the function `mset_set` in Isabelle. Thus, we can define the estimate as follows:

```
definition estimate 𝒜 ≡ λ(P,L).
  let xs = (SOME xs. xs <~~~> L) and
      ys = (SOME ys. ys <~~~> Σ 𝒜 × P - L) in
  ∑s←xs. card(preds 𝒜 (fst s) (snd s)) * log(card(snd s)) +
  ∑s←ys. card(preds 𝒜 (fst s) (snd s)) * log(card(snd s)/2)
```

Remark 3. This definition makes use of the non-deterministic choice operator `SOME` of Isabelle to choose a permutation of L and $\Sigma \times P - L$ respectively. Since we are working with finite automata, such permutations always exist.

[8] The operator `*m` lifts multiplication to cost expressions.

[9] A list ℓ permutes a set S, which we denote by ℓ `<~~~>` S, iff `mset` ℓ `=` `mset_set` S.

Simplifying the estimate: Unfortunately, such a definition makes the proofs very difficult. For a state transition $(P, L) \to (P', L')$, we would have to examine all cases for all blocks whether they are splitters in L or whether they are split by some other splitter or whether there are new splitters created from this splitter in L'. This would be very tedious to formalize, but here is a sketch of some ideas.

Splitters of the workset can be uniquely characterized thanks to "a-q-splitters". It is defined as follows for a state (P, L):

Definition 3. Let $a \in \Sigma$ and $q \in \mathcal{Q}$. Let B be the unique block of the partition P containing q. If $(a, B) \in L$, it is the **a-q-splitter**. Otherwise, there is no a-q-splitter.

This can be achieved in Isabelle using the option type. Although this is a good characterization of splitters, it is not easy to work with it. For example, if we want to show that the number of splitters decreases, we would have to show that the number of a-q-splitters decreases for any $a \in \Sigma$ and any $q \in \mathcal{Q}$. This is rather tricky to formalize.

In order to simplify the expression of the estimate (1), we can remove the halving factor in the second term, which allows both term to be gathered together under one sum. We obtain an upper bound for the previous expression and the difference between the two is "small":

$$\begin{aligned} T &\leq \theta \cdot \left(\sum_{(a,C) \in \mathcal{L}} |\overset{\leftrightarrow a}{C}| \log |C| + \sum_{(a,B) \in (\Sigma \times \mathcal{P}) \setminus \mathcal{L}} |\overset{\leftrightarrow a}{B}| \log |B| \right) \\ &= \theta \sum_{(a,C) \in \Sigma \times \mathcal{P}} |\overset{\leftrightarrow a}{C}| \log |C| \end{aligned}$$

Thus, we work with the following simplified estimate:

$$\hat{T} := \theta \sum_{(a,C) \in \Sigma \times \mathcal{P}} |\overset{\leftrightarrow a}{C}| \log |C| \tag{2}$$

The definition in Isabelle is also simpler:

```
definition estimate 𝒜 ≡ λ(P,L).
  let xs = (SOME xs. xs <~~~> Σ 𝒜 × P) in
  ∑s←xs. card(preds 𝒜 (fst s) (snd s)) * log(card(snd s))
```

References

[EB23] Javier Esparza and Michael Blondin. *Automata Theory: An Algorithmic Approach*. 2023.

[Hop71] John E. Hopcroft. *An n Log n Algorithm for Minimizing States in a Finite Automaton*. Stanford University, Stanford, CA, USA, 1971.

[LT12] Peter Lammich and Thomas Tuerk. Applying data refinement for monadic programs to hopcroft's algorithm. pages 166–182, 08 2012.