# Grading Student Solutions for Automata

RUBEN HANNINK, University of Twente, The Netherlands

Even though multiple solutions exist for grading automata, these solutions have some limitations which can negatively impact the grade of students. This is because most algorithms either check whether a solution given by a student is entirely correct, or entirely incorrect. There are already existing solutions for grading automata with partial grades, however these are extremely limited in which actions can be executed on an automaton. The goal of this study is to determine correction rules that are fairer to the student and does not punish a student for continuing with the same mistake. Together with creating a new algorithm to grade the student solutions, which provides transparency about how the algorithm determined the grade. This would allow teachers to start using automated grading of automata, significantly decreasing the workload. In this paper 11 corrections are proposed, which can be used to transform an automaton until it is a correct solution. For these rules, fairness towards the student was kept in mind. Together with these corrections a specification for an algorithm to apply these corrections and use that to grade solutions has been given. Unfortunately, it became apparent that the given algorithm in its current form is not very performant and not feasible to be used on a lot of solutions, this results from the substantial number of branches that the algorithm has to traverse. However, with more research it can be expected to achieve a better performance. With this, the paper provides a valuable first step into making a more general algorithm for grading student solutions using partial grades.

Additional Key Words and Phrases: Grading, Automata, Distance-based Algorithms

## 1 INTRODUCTION

As part of the curriculum for Technical Computer Science at the University of Twente there is a module which has a subject called "Languages and Machines" (L&M). This module focuses on Languages, Regular Expressions, and Automata. For this research we will only be focusing on the automata section. During L&M students learn how to create an automaton representing a language, usually from a regular expression, and how to write a regular expression that represents the same language as an automaton.

During the test for Languages and Machines students will be asked various kinds of questions related to automata. In most of the exams the same kind of questions are asked, testing whether the student has specific knowledge about automata and can apply that knowledge. Below are a couple of examples of what knowledge or skills the questions try to assess.

- (Q1) The student can be presented with an automaton and needs to answer several questions on whether certain words would result in an accepting state.
- (Q2) The student can be given an automaton for which the student will need to write down the regular expression for.
- (Q3) The student can be presented with an automate for which only the language, transition arrows and states are given. The student will need to fill out what the goes in the blank squares (Figure 1).

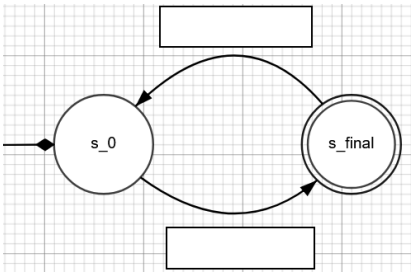


Fig. 1. Example test question, automata with transition arrows and states only for the language L(E), with regular expression E = (ab)*.

The goal of these questions is to evaluate the student's understanding of automata by testing whether they can read and change any automaton they are given. To ensure the answer is not a lucky guess, the student also needs to show how they arrived at the answer for all questions. The way students need to show their steps ranges from giving the sequence of states that were passed to get to an accepting state, to showing intermediary steps when transforming an automaton into a regular expression. However, this focuses mostly on whether students can read automata, there is less of a focus on whether students can draw automata. Q3 has the most of focus on drawing automata of all types of questions, however the student is helped by the fact they are already given the states and transition arrows (Figure 1) instead of needing to start from nothing.

This is because there are some difficulties with asking students to draw these automata. The university already has a tool which can be used during digital examinations in which students can draw automata: UTML [8]. The difficulty does not come from how to draw the automata during an exam. The problem is grading these solutions. For a small automaton it is easy, but as the automaton gets bigger it becomes a very time-consuming task to grade them all manually, especially when partial grades should be given for solutions that are close to being correct.

## 2 PROBLEM

Checking whether a solution is correct is already possible, even if the drawn automaton differs, which will be discussed in more detail in Section 3. However, as discussed in the introduction, the difficulty comes from the need for partial grades. For this we need to be able to find the minimal number of fixes that need to be applied to an automaton to get it to a correct solution. Before we can do this, the different fixes that can be made will need to be established first. After this we can figure out a procedural way of checking the minimal amount of these previously defined corrections to get to a correct solution.

*TScIT 43, July 4, 2025, Enschede, The Netherlands*

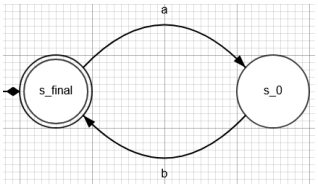


Fig. 2. Correct automaton for L(E) with E = (ab)*

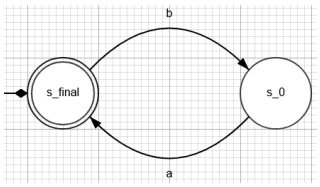


Fig. 3. Incorrect automaton for L(E) with E = (ab)*, with the 2 transitions swapped.

### 2.1 Research Question

To achieve this goal the following research question (RQ) will need to be answered:

*How to systematically determine the minimum number of fixes required by a student solution for automata to arrive at a correct solution for the purposes of grading.*

This can be split into 2 sub research questions (SRQ):

**(SRQ.1)** *What is the set of corrections needed to convert any incorrect solution into a correct solution, considering the types of mistakes students make?*

**(SRQ.2)** *How to systematically apply these corrections and determine the minimal number of corrections needed to achieve a correct solution?*

## 3 RELATED WORKS

In the field of grading automata there are already some related works solving these questions. However, our goal is slightly different and each of the already existing works has some limitations when trying to answer our research questions.

Firstly, there is already research done to check the correctness of an automata with the ability to give simple feedback [7], however this does not give enough information to grade the solution using partial grades.

The second solution found is from 2013 [1]. This research focused on deterministic finite automata (DFA), which used several rules to determine a "distance" between two automata. This distance is equal to the minimum number of corrections that needed to be made according to the rules, which can then be used to grade the solution. This has a lot of similarities to this research. However, there will be a significant difference between the rules that were considered during their research, and which rules will be considered during our research. The rules proposed in this paper do not account for mistakes being related to each other. An example of this is mistakenly swapping two transitions around, this can be seen as a single mistake, however with the rules from this paper, they are counted as 2 separate mistakes. Using the same example language as in Figure 1, from the regular expression `E = (ab)*`. The solution in Figure 2 is correct, however the transitions are swapped in Figure 3. Where this solution would determine there to be 2 errors, 1 for each of the arrows. A teacher may disagree and say that it is only 1 error in total. There is a need for more rules than in given in this solution. Additionally, the algorithm described in this paper focuses on automata written in the `MOSEL` language, not a general solution to the problem.

To know whether 2 automata represent the language, we need to have a way of systematically comparing the languages of 2 automata. Hopcroft and Karp [5] have already made an algorithm that solves this problem. There is, however, not an algorithm defined in the paper where the empty word ($\lambda$) can be used in the automaton.

There is already an existing algorithm that can convert an NFA with the empty word (NFA-$\lambda$) to a DFA. This algorithm was made by KJai Salomaa and Sheng Yu [6]. The algorithm works by walking over the NFA and constructing a new DFA representing the same language as the original NFA-$\lambda$.

With this tool it would be possible to use the algorithm by Hopcroft and Karp [5] in the grading algorithm.

## 4 METHODOLOGY

The related works discussed in Section 3 will be used as a starting point for answering the research question and both sub research questions. The sub research questions have been written in such a way that, when both are answered, the main research question is also answered.

### 4.1 Answering SRQ1

To answer this question, the different corrections the algorithm can execute need to be established. Rajeev Alur et al already established a couple of these corrections [1]. The goal will be to expand this set of rules to add new rules that make sure some actions are combined. These rules will need to be in favor of the students, making sure it will reduce the number of corrections needed. Using the example in Figure 3 again, these rules will need to make sure it only counts one correction, not the 2 corrections which would be required when using the algorithm by Rajeev Alur et al [1]. For this, existing exercises from L&M will be used, where we will reason on which mistakes students can make and why certain actions should be combined.

### 4.2 Answering SRQ2

Like the algorithm by Rajeev Alur et al [1], a distance-based algorithm will be made to answer the second research question. Here the focus lies on making sure the new algorithm is a general solution to the problem, and transparent about what changes are needed in the minimal solution, so the teacher can still determine some rules that should be combined or ignored. This transparency is important from an ethical point of view [3], as this makes it possible to detect mistakes the algorithm made in grading the solution.

As part of the goal for answering this sub research question, a small demo using this algorithm will be implemented.

## 5 IMPLEMENTATION

### 5.1 Correction rules

When receiving a student solution, we can, in this case, only assume it is an NFA-$\lambda$. Any invalid transitions should be dropped when loading in the automaton.

Below are the different corrections which can be applied to an automaton:

**(C.1)** **State insertion:** Create a new state $q$.

**(C.2)** **State deletion:** Delete a state $q$ and all transitions associated with it.

**(C.3)** **Relabel state:** For a state, add it to, or remove it from the list of accepting states.
**(C.4)** **Relabel all states:** For all accepting states, make them non-accepting, and for all non-accepting states, make them accepting.
**(C.5)** **Transition redirection:** Redirect a state transition to a new target. (Change a transition $\delta(q, a) = q'$ to $\delta(q, a) = q''$ with $q' \neq q''$)
**(C.6)** **Transition flip:** Flip a single transition around. (Given a transition $\delta(q, a) = q'$) change it to $\delta(q', a) = q$)
**(C.7)** **State transitions flip:** Flip the transition between two states. (For the states $q, q' \in \Sigma$, for all $a \in \Sigma$, change $\delta(q, a) = q'$ to $\delta(q', a) = q$ and/or $\delta(q', a) = q$ to $\delta(q, a) = q'$, if they exist)
**(C.8)** **Transition insertion:** Insert a new transition.
**(C.9)** **Transition removal:** Remove a transition.
**(C.10)** **Relabel transition:** Change a transition $\delta(q, a) = q'$ to $\delta(q, b) = q'$ where $a \neq b$.
**(C.11)** **Change start transition:** Change the start transition to indicate a different entry point of the automaton.

Within this list C.1, C.3, and C.5 were taken from the proposed differences between automata in Alur et al [1]. The state insertion (C.1) was defined as "insert a new disconnected state $q$, with $\delta(q, a) = q$ for every $a \in \Sigma$" [1] in the original paper. However, C.1, as stated above, does not add these transitions, as in our case redirecting a transition in order to point it to the correct destination has the same impact on the grade as inserting a new transition at a later point.

*5.1.1 State corrections.* The first step is to let the algorithm create and delete states (C.1, C.2), allowing the student to forget a state, or write unnecessary states. The state deletion will also delete any incoming and outgoing transitions in the same action to prevent transitions that go nowhere.

States within an automaton have the following properties: `Label` and `IsAccepting`. The label is the text displayed on the state, which we will ignore for the corrections, as the goal is to make sure the automaton accepts the right language.

For the `IsAccepting` property some corrections might be needed. This property indicates whether this is an accepting or non-accepting state within the automaton. Changing this property for a single state (C.3) will be needed if the student mistakenly puts an additional accepting state or forgets to make a state accepting. On the other hand, the student may mistakenly place accepting states where they intended to place non-accepting states, and vice versa. Because of this a correction is added for changing the property for *all* states (C.4), meaning the algorithm will only need a singular correction to correct it.

*5.1.2 Transition corrections.* Similar to the states, creating and removing transitions is necessary (C.8, C.9). Unlike with states, these corrections have no side effects.

Transitions have the following properties: `StartState`, `EndState`, `AcceptingSymbol`. For correcting the transitions, it should be possible to change small mistakes, but the transition should not be disconnected from both states and moved to an entirely different part of the automaton, it should always stay connected to at least one of the two states. In order to achieve a transition away from both states two operations should be necessary (C.8 and C.9) as this indicates a bigger error in the automaton.

The first transition correction (C.5) focuses on changing the `EndState` of the transition. A student may make a mistake in creating the automaton, causing an incorrect path when a transition is pointed to the wrong state. Similarly flipping transitions (C.6) focuses on transitions that have been drawn the wrong way. However, a transition can only be flipped if the automaton remains a valid DFA. This same rule applies to the flipping of all transitions between two states (C.7), this ensures that if both transitions have been drawn the wrong way, a single correction can fix it.

The last correction that can be applied to transitions is changing the symbol it consumes (C.10).

With all these corrections combined the algorithm has full control over changing the transitions and some of the smaller mistakes can be corrected in a single correction.

*5.1.3 Start transition correction.* The last property the algorithm needs to be able to change is which symbol is the starting symbol (C.11). This will ensure that an automaton where the student accidentally misplaced the start transition does not need to be morphed entirely to fit the language.

## 5.2 Algorithm

This section focuses on the implementation details of the algorithm and the design considerations that were made.

The algorithm described is a distance algorithm, attempting to find the smallest distance between two automata. Dijkstra's algorithm [2] was considered as an approach. The problem with taking Dijkstra's algorithm is twofold.

Firstly, the algorithm would need to store *all* traversed automata. Although each automaton is relatively small, the tree that needs to be searched can become enormous, especially for bigger automata or questions that have the possibility of a higher score. For students that receive partial grades the impact may be minimal, but for students who receive 0 points the algorithm has to traverse all possibilities.

Secondly Dijkstra's algorithm would require the algorithm to detect when 2 automata are equal. This is not detecting when the languages of 2 automata are equal, as described in Section 5.2.2, but needs to verify that 2 automata are the same graphs. This requires an additional comparison algorithm to be written and researched, together with the performance impact of checking each automaton against all the other, already discovered, automata, which gets more resource intensive as more options are explored.

For these reasons, the decision was made against using Dijkstra's algorithm, instead using a custom solution as described in this section. For the algorithm, a Python 3.12 implementation was made [4].

*5.2.1 Automaton Representation.* For the automata representation Python classes were used. The following classes were created:

- `Automaton` -> Represents the automaton with all its states and transitions
- `Transition` -> Represents a transition within an automaton
- `State` -> Represents a state within an automaton

- `Symbol` -> Represents a symbol that can be added to a transition

The symbol class was created out of necessity for the empty word. The symbol class can either be given a character or `None` when created, when given a character it represents the symbol of that character, when given `None` it represents the empty word.

In the Python representation of the automaton multiple transitions with the same start and end node can be combined. For this reason, the `Transition` class has a list of `Symbols` instead of just one. Whenever a transition is created, it will first need to check if a transition from the start to the end node already exists, if it does it will need to combine the two transitions by adding the symbol to the existing transition. Only if no transition exists yet does it create a new one.

If a transition with a symbol already exists from the start to the given end node, no changes will be made, and the method should indicate this in its return (in the example implementation this is done by returning `None` instead of the `Transition`). Corrections will need to take this into account, most corrections related to transitions will need to remove the created transitions afterwards, however if no transition was created it should not remove a transition (this would cause side-effects to the undoing of the correction otherwise). In most cases this will make the correction invalid (Section 5.2.3).

The automaton should keep track of a list of "initial states". These starting transitions should not be represented by actual transitions, instead the node a starting transition points to, should be included in the initial states list.

*5.2.2 NFA-$\lambda$ to DFA and Equivalence.* In order to know whether the algorithm has reached a point where the automaton, with corrections applied, represents the same language as a correct solution we will be using Hopcroft and Karp's algorithm for comparing DFAs [5] combined with Salomaa and Yu's algorithm for converting NFA-$\lambda$ into DFAs [6].

We only want to run the NFA-$\lambda$ to DFA converter when either of the 2 automata that needs to be checked is a DFA. For this a simple checker needs to be added, to see whether an automaton is a DFA or not. This can be done by checking if the empty word is not part of the symbols within the automaton and that for each state in the automaton, there exists at most 1 transition for each symbol.

As the algorithm has to repeatedly compare against the same correct automaton, it would waste resources if we had to change the correct automaton from an NFA-$\lambda$ to DFA each time the 2 automata are compared. Because of this a `Validator` class was made. This class stores a reference to the correct automaton if it is a DFA, or to the converted automaton if it is not a DFA. This way the conversion only has to only happen once for the correct solution.

*5.2.3 Corrections.* In Section 5.1 we laid out the different corrections that are needed to transform an automaton into one representing the same language. To apply these, we will need to implement their functionality. For each of the corrections we will need to make 2 functions, an application function, and an undo function.

Alternatively, a clone of the automaton can be kept from before the correction is applied, reverting to this clone will be the same as undoing. The decision for the approach including an undo function, instead of using clones, was made as most of the corrections can be undone relatively quickly compared to creating an entire copy of the automaton. This also means that the speed of undoing the correction is not dependent on the size, as most references can be directly gotten using list indices or dictionaries in the automaton.

To help with keeping track of which corrections are applied on an automaton in the future, we will be needing to store each of the corrections as a class to ensure some metadata can be stored. This also doubles as a way to provide transparency, as when a final solution has been found, the teacher will be able to check whether the applied corrections are right.

We assume that the application of corrections works as a stack. When applying a correction, it is added to the top of the stack and can only be undone (and removed) once all elements that were placed above it are undone (and removed). Taking the State insertion (C.1) as an example, when trying to undo the correction, we assume any corrections applied after the application have already been undone, therefore there cannot be any transitions attached to the state and the state can easily be removed by removing it from the state list of the automaton.

Below is a list detailing the implementation of each of the corrections. All the correctors require the automaton as an input, which has been left out in the arguments listed below to prevent repetition, the other arguments are listed with their types. Beside the arguments, and the application and undo instructions, there is also a determination for when a correction is "Valid". When a correction is not valid it indicates it was not possible to apply the correction, it is therefore also not needed to undo invalid corrections.

The arguments for the provided Python implementation [4] may differ slightly as states are gotten by their state ID instead of a direct reference.

**State insertion (C.1):**
*Arguments:* accepting (`boolean`).
*Application:* Adds a new state to the automaton. The "accepting" parameter determines whether the newly created state is an accepting state.
*Undo:* Removes the state from the state list of the automaton.
*Valid:* Always valid.

**State deletion (C.2):**
*Arguments:* state (`State`).
*Application:* Remove the `State` from the states list of the `Automaton`. Remove all transitions from the transition list of the `Automaton`, and all references that are kept to these transitions in other `States`.
*Undo:* Will place the `State` back into the state list, also placing back all transitions into the transition list of the `Automaton`. Additionally, all references that were kept in other states are added back in.
*Valid:* Always valid.

**Relabel state (C.3):**
*Arguments:* state (`State`).
*Application:* Change whether the `State` is accepting or not, this will toggle the `IsAccepting` boolean.

*Undo:* Same as application, flipping the `IsAccepting` boolean back to the value it had before application.
*Valid:* Always valid.

**Relabel all states (C.4):**
*Arguments:* no additional arguments.
*Application:* For each state in the automaton, toggle the `IsAccepting` value. This is the same as applying C.3 to all states.
*Undo:* Same as application, changing all the `IsAccepting` values back to the value it had before the application.
*Valid:* Always valid.

**Transition redirection (C.5):**
*Arguments:* start (`State`), old end (`State`), new end (`State`), symbol (`Symbol`).
*Application:* Find the transition that goes from the start to the end, using the given symbol. Then change the end node of this transition to be the new end state.
*Undo:* Change the end state of the transition back to the old end state.
*Valid:* Valid if a transition using the given symbol exists from the start to the end state, *and* the end state of this transition is not the new end state.

**Transition flip (C.6):**
*Arguments:* start (`State`), end (`State`), symbol (`Symbol`).
*Application:* Find an outgoing transition from the start state to the end state, using the given symbol. Then swap the start and end states of the transition.
*Undo:* Flip the transition back, switching the start and end states back around.
*Valid:* Valid if a transition using the given symbol exists from the start state to the end state, *and* the flipped transition does not exist yet (as we cannot have the same transition twice).

**State transition flip (C.7):**
*Arguments:* state a (`State`), state b (`State`).
*Application:* For all transitions between the given state a and b, it will swap the start and end nodes. So, for each state starting at a and ending at b, it will start at b and end at a, and vice versa.
*Undo:* Flip the transition back, switching the start and end states back around.
*Valid:* Valid if there are any transitions between the two given states.

**Transition insertion (C.8):**
*Arguments:* start (`State`), end (`State`), symbol (`Symbol`).
*Application:* Create a new transition going from the given start to the given end with the given symbol.
*Undo:* Remove the transition created in the application.
*Valid:* Valid if no transition going from the state to the end state already exists with the given symbol.

**Transition removal (C.9):**
*Arguments:* start (`State`), end (`State`), symbol (`Symbol`).
*Application:* Find and remove the transition going from the start state to the end state, using the given symbol.
*Undo:* Insert the transition again by creating a new transition from the start state, to the end state, using the given symbol.
*Valid:* Valid if the transition that needs to be removed could be found.

**Relabel transition (C.10):**
*Arguments:* start (`State`), end (`State`), old (`Symbol`), new (`Symbol`).
*Application:* Find and remove the transition going from the start state to the end state, using the given symbol. Then create a new transition with the new symbol, going from the start state to the end state.
*Undo:* Remove the newly created transition using the new symbol and insert the transition with the old symbol again by creating a new transition from the start state to the end state.
*Valid:* Valid if an outgoing transition with the given old symbol exists for the start and end states, *and* no transition going from the start state to the end state, already exists using the new symbol.

**Change start transition (C.11):**
*Arguments:* state (`State`).
*Application:* Copy and store the list of current start states. After which the list will get cleared and the given state will be added into the list, making it the only start state.
*Undo:* Replace the current start states of the automaton with the stored list, placing the start states from before the application back.
*Valid:* Valid if the given state is not already the only start state.

*5.2.4 Correctors.* To check how much corrections are needed the corrector algorithm will need to go over all possibilities for each of the corrections and check whether it can find a correct automaton. For this the algorithm requires 3 arguments.

- A correct automaton for the language.
- The student solution (incorrect automaton).
- The maximum number of points that can be awarded.

For each of the corrections from Section 5.2.3 a corrector will be made. Each of these correctors will apply all possible corrections for their type, after which they will try and continue to the next application. When referring to applying all possible corrections, we mean applying all possibilities of all the single corrections, not applying multiple corrections to the automaton.

The algorithm works by brute forcing all combinations of corrections. First it will check if the given solution is already correct, if it is full points will be awarded. Otherwise, it will start with a depth of 1, meaning it will try all possible corrections, and for each check whether the automaton is correct after applying. If it is, that will be the solution, otherwise the correction is undone, and the algorithm continues to the next.

After checking all possibilities for a depth of 1, all possibilities for a depth of 2 are checked. Here the algorithm will go over all possible corrections and attempt to add a second correction, by going over all possible corrections again, this attempts all combinations of 2 corrections. This depth will be increased until it has reached the $max\_points$, for which it will not run. This is done as running for a depth of $max\_points$ will result in a score of 0, which will also be the result if the algorithm could not find a solution, so there is no need to run this depth.

This is comparable to brute forcing a set of numbers. First attempting all possibilities for the first number (e.g. 0000, 1000 ... 9000). Then, for each possibility of the second number, we try all the possibilities of the first again (e.g. 0100, 1100 ... 9100, 0200, 1200 ... 9900) and continue to increase the depth for the length of the desired number.

For the implementation of the algorithm this depth can be seen as a chain of actions. Here we create the following 2 actions:

- `CheckAnswer` -> Check whether the current automaton is correct, if it is, the solution has been found.
- `RunAllCorrectors` -> Run all possible corrections, after each it will call the next action (Algorithm 1).

**Algorithm 1** RunAllCorrectors action

```
next_action ← the action to run after this
first ← whether this is the first action in the chain
for corrector ∈ correctors do
    if first is False and corrector is RelabelAllStatesCorrector then
        continue ▷ Skip the RelabelAllStatesCorrector if we are not the outermost action
    end if
    run corrector, giving the next_action as its next action
end for
```

Here the `RunAllCorrectors` will form a chain of instances, with a `CheckAnswer` instance at the back. When running over all the correctors, each of the correctors follows Algorithm 2, attempting all possibilities and then running the next action. In both Algorithm 1 and Algorithm 2, the `next_action` is either an `RunAllCorrectors` or `CheckAnswer` instance, this way a chain of `RunAllCorrectors` instances can be formed with a `CheckAnswer` instance at the end. The length of the `RunAllCorrectors` chain will represent the depth, each time when increasing the depth, the chain will be extended with another `RunAllCorrectors` instance (Algorithm 3).

**Algorithm 2** Corrector

```
next_action ← action given in arguments
for each possible correction, correction do
    Try to apply correction
    if correction is valid then
        run the next_action, marking it as not the first in the chain
        if a valid solution has been found then
            break ▷ Stop execution if a solution was found
        end if
        Undo correction
    end if
end for
```

The `Corrector` for relabeling all states should only be run once on an automaton, applying it a second time will always undo its previous application. Because of this Algorithm 1 has a check, when it is not the outermost the `RelabelAllStatesCorrector` will not be run, this means it can only be called once in the first action of the chain.

**Algorithm 3** Main algorithm

```
max_points ← the maximum amount of points that can be awarded
if automaton is correct then
    return max_points
end if
action ← new CheckAnswer instance
action ← new RunAllCorrectors instance, using action as the next_action
depth ← 1
/*We now have a chain of length 1 , where after the RunAllCorrectors action the next action is a CheckAnswer action*/
while depth < max_points do
    run the action, marking it as the first in the chain
    if correct solution was found then
        break ▷ Quit the while loop early
    end if
    action ← new RunAllCorrectors instance, using action as the next_action
    depth ← depth + 1
end while
return max_points − depth
```

## 6 RESULTS AND VALIDATION

To validate whether the implemented algorithms, test cases are provided in the Python implementation [4].

The first set of test cases focuses on running the algorithm where only 1 correction is necessary to make the languages of the automata the same. This is intended to show the algorithm can apply all the corrections and check for possible side effects of applying and undoing the corrections. As these tests only need to explore a depth of 1 before finding the correct solution, they all complete quickly (>0.01 seconds).

The second set of test cases focuses on evaluating the performance of the algorithm. These test cases are created such that the student solution will result in 0 points, showing the time it takes to explore all the possibilities up to a certain depth. For this, 4 different pairs of automata were created (a correct automaton and an incorrect automaton), with different sizes. Each of the pairs is being run through the algorithm 3 times, every time with a different number of max points, running the algorithms to different depths. Here the max points of 1 is skipped, as this would only check whether the initially given automaton is correct, not attempt to apply a correction as any solution found after applying a correction would also result in 0.

The results of these timed tests are shown in Table 1. Here the size columns contain a tuple `(state count, transition count)` for the correct and incorrect solutions. In the automaton representation proposed in Section 5.2.1 the transitions were combined, here for the transition count, all individual symbols of each transition in the automaton are counted as a transition. Each of the test cases was given a limit of 10 minutes after which it would be marked as "Timed out".

Table 1. Table of time results of the timing tests

| Test case | Correct size | Incorrect size | Max points | Time (s) |
|---|---|---|---|---|
| Case 1 | (5, 9) | (4, 9) | 2 | 0.12 |
| | | | 3 | 0.98 |
| | | | 4 | 108.42 |
| Case 2 | (7, 15) | (6, 15) | 2 | 0.13 |
| | | | 3 | 3.52 |
| | | | 4 | Timed out |
| Case 3 | (7, 18) | (6, 18) | 2 | 0.14 |
| | | | 3 | 8.53 |
| | | | 4 | Timed out |
| Case 4 | (8, 21) | (8, 21) | 2 | 0.16 |
| | | | 3 | 32.58 |
| | | | 4 | Timed out |

When looking at the results in Table 1 we can look at the different times gotten for each of the cases, depending on the maximum amount of points, or we can look at the difference in time for the same amount of points, but between the different cases and automaton sizes.

First, looking at the difference in time depending on the max points of a run, it becomes clear that the time taken by the algorithm greatly increases when the depth increases. Taking the example of brute-forcing a 4 digit code again, we can show that attempting everything for only the first digit (depth of 1) only gives us 10 possibilities, however once we also need to check a depth of 2 we need to check all possible first digits, for each of the possible second digits, so we are already up to $10 * 10 = 100$ possibilities. A similar things happens while running the algorithm, at first only the possibilities for one correction are applied, but after that for each of the possible corrections, we need to go over all possible correction again in order to test all possibilities of 2 corrections, already significantly increasing the amount of possibilities.

Looking at the various times between the test cases, we can compare the time for a given depth, to the given depth of the other cases. This means we are only comparing the impact the size of the automaton has on the time it takes. Here the increase in time is most notable when taking the results for the max points of 3. With only a couple of additional states or transition the time is more than doubled when comparing each test case with the next.

Case 2 and 3 were also specifically made to only change the number of transitions between the cases. Here we can see that even a slight increase in transitions can already lead to a significant increase in time.

This increase in time also only depended on the size of the incorrect solution, as the list of corrections which can be applied is based on the contents of the incorrect solution.

With this stark difference in time between the automata, and several test cases exceeding 10 minutes with the max points set to 4, it is not viable to use this algorithm, in its current state, on many students.

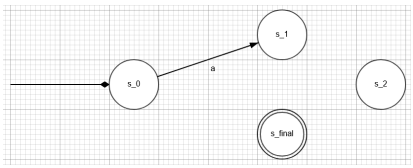


Fig. 4. Incorrect automaton for the language L(E) with E = (a)

## 6.1 Performance improvements

The algorithm was made with a few performance improvements in mind. Unfortunately, due to time constraints, it was not possible to add these improvements to the implementation of the algorithm.

The first improvement that can be made is by choosing which corrections get applied. Currently the algorithm for each of the corrections loops over *all* possibilities, even if they result in invalid corrections. The performance impact of an invalid correction is not remarkably high, however in some cases the number of invalid corrections can be exceptionally large, even the small impact of checking if it is correct will impact performance. An alternative would be to already include some rules on which corrections are attempted. As there still needs to be some checks, the performance increase of this improvement will most likely be small.

An improvement which could be a significant performance increase is reducing the number of branches that need to be explored. Similarly to how the `RelabelAllStatesCorrector` can only be applied once to an automaton, as calling it multiple times undoes its effect, the branches could be further reduced if such rules were applied to all corrections. In these cases, it should still run the corrector, but the corrector should make sure no corrections are applied that undo the work of any previous corrections. The information about corrections that are applied is already stored in a list, in the Python implementation, this could then be used to determine which corrections should not be applied later.

As an example, take Figure 4, which shows an incorrect automaton for the language only accepting the word "a". When applying corrections, we can first change the transition to end at "s_2", after which we can change it again to end at "s_final", making it a correct solution. This means 2 corrections must be applied. However, we can also change the transition to end at "s_final", only needing 1 correction to get to a correct solution. Redirecting the same transition twice does not make sense, as that could have been a single correction, however with the current algorithm this happens a lot. Removing these can greatly reduce the number of branches, thus creating a significant performance improvement.

Additionally, there will be added value in making an implementation based on Dijkstra's algorithm [2]. In Section 5.2 the decision was made to not use Dijkstra's algorithm due to the performance impact of needing to check an automaton against all, already discovered, automata when applying a new correction. However, this combining will also have the effect of merging branches with each other. The question that remains is, will this reduce the number of branches enough, so with the extra processing required to check against all discovered automata, it is still more performant.

## 7 CONCLUSIONS

Within this paper we presented an algorithm that can be used for grading student solutions for automata and provided a Python implementation. The 2 sub-research questions asked in the paper have both been answered, therefore the research question itself has been answered.

The first question, (SRQ.1) *What is the set of corrections needed to convert any incorrect solution, into a correct solution, considering the types of mistakes students make?* It has been answered using a list of 11 corrections that can be applied to transform any DFA into any other DFA. For the corrections in this list fairness towards the student has been considered, making sure the student does not get unnecessarily punished for mistakes when a student continues with a previously made mistake (e.g. swapping transitions around).

Answering the second question was more difficult, (SRQ.2) *How to systematically apply these corrections and determine the minimal number of corrections needed to achieve a correct solution?* The algorithm described in this paper can grade student solutions by determining the minimal number of corrections, thus answering the second question. Unfortunately, the algorithm in its current state is extremely limited by its performance, it is not feasible to use the algorithm on a large number of solutions or on larger automata. Although, with additional research into the proposed performance improvements, or other performance improvements, the solution may become viable.

In the end, this paper has provided a valuable first step into making a more general algorithm for grading automata by determining the corrections that can be applied and giving an initial algorithm which can grade automata.

## REFERENCES


[1] R. Alur, L. D'Antoni, S. Gulwani, and D. Kini. 2013. Automated Grading of DFA constructions. *IJCAI '13 Proceedings of the Twenty-Third international joint conference on Artificial Intelligence* (2013), 1976–1982. https://www.microsoft.com/en-us/research/publication/automated-grading-dfa-constructions/

[2] E. W. Dijkstra. 1959. *A note on two problems in connexion with graphs.* Springer Nature Switzerland, Cham, 269––271. https://doi.org/10.1007/BF01386390

[3] A. Farazouli. 2024. *Automation and Assessment: Exploring Ethical Issues of Automated Grading Systems from a Relational Ethics Approach.* Springer Nature Switzerland, 209–226. https://doi.org/10.1007/978-3-031-58622-4_12

[4] R. Hannink. 2025. Python implementation for grading student solutions for automata. https://github.com/RHannink01/AutomataGrader. (Last accessed 29 june 2025).

[5] J. E. Hopcroft and R. M. Karp. 1971. A linear algorithm for testing equivalence of finite automata. *Technical Report 114* (1971).

[6] S. Kai and Y. Sheng. 1997. NFA to DFA transformation for finite languages. In *Automata Implementation.* Springer Berlin Heidelberg, 149–158. https://doi.org/10.1007/3-540-63174-7_12

[7] A. Kumar, A. Walter, and P. Manolios. 2023. *Automated Grading of Automata with ACL2s.* Vol. 375. Open Publishing Association, 77–91. https://doi.org/10.4204/eptcs.375.7

[8] Utwente UML 2021. UTML. https://labs.apps.utwente.nl/apps/utml.html. Public live version: https://utml-staging.apps.utwente.nl/ (Last accessed 26 june 2025).