

Automata Algorithm Module

With Web Integration for Teaching Theoretical
Computer Science



Public version of bachelor defense slides

By: Thomas Helboe Poulsen and Janus Dalsgaard Schnabl

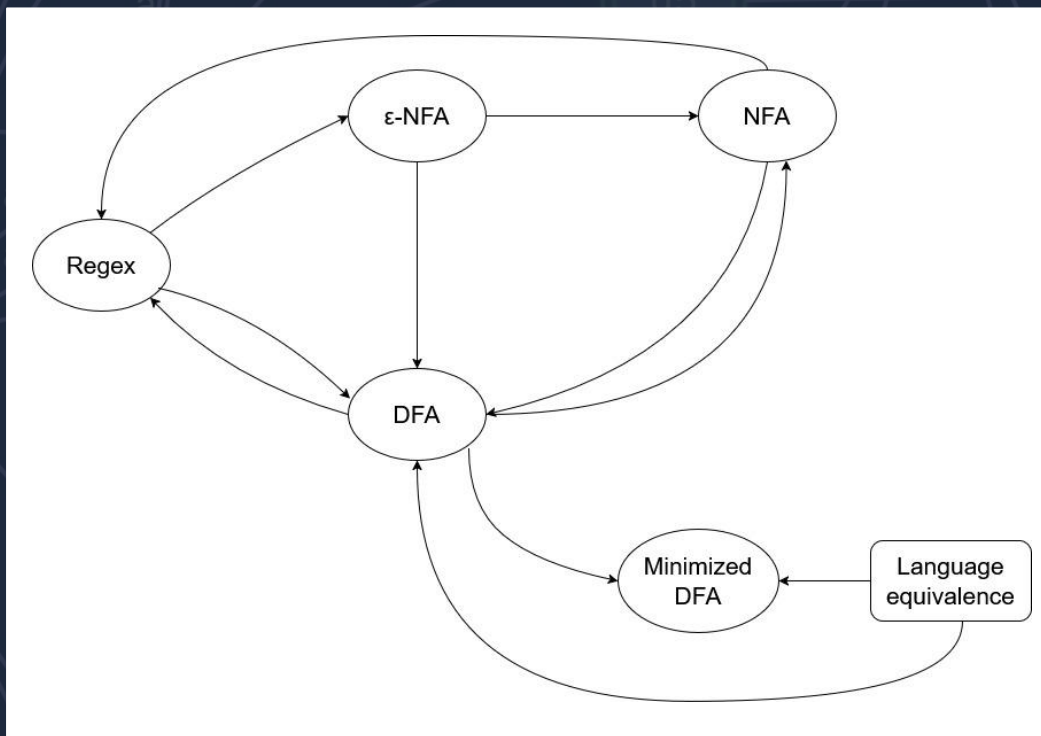
Checkr: What is it?

- A tool used for learning about abstract computer science
- Students implement algorithms in F# (starters)
- See reference solution and get feedback
- Frontend is called inspectify
- Submissions can be evaluated using course-defined test inputs

Checkr: What material is covered?

- Parser
- Compiler
- Interpreter
- Signs Analyzer
- Information Flow Analysis

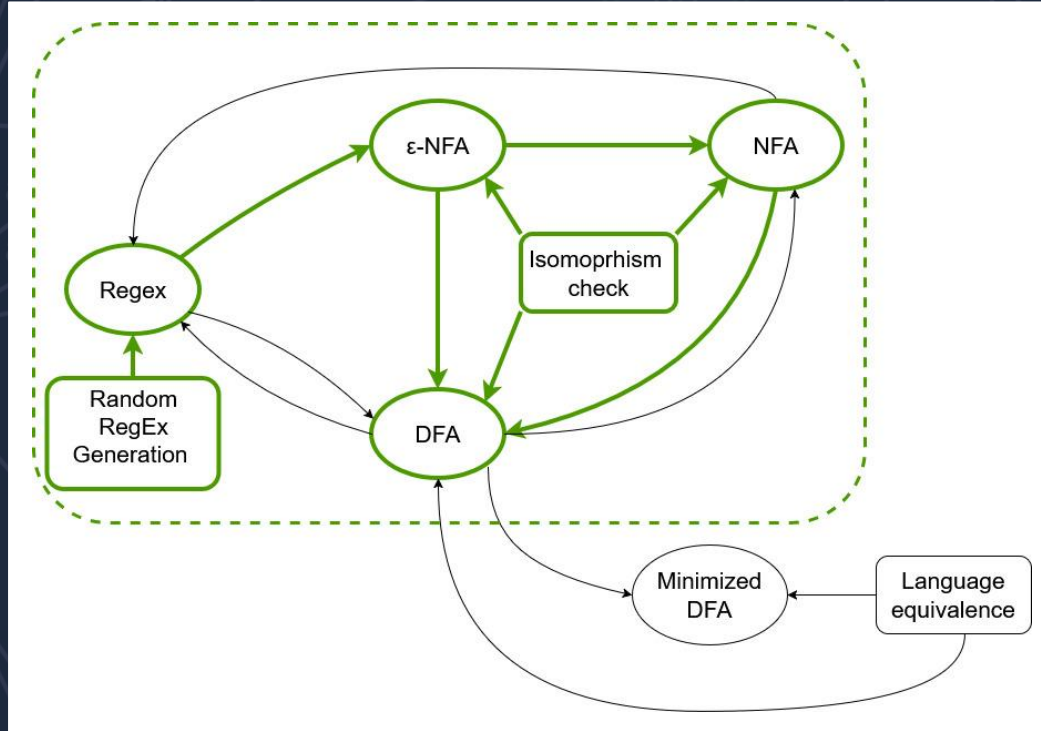
Checkr: What is missing?



Checkr: Limitations of existing tools

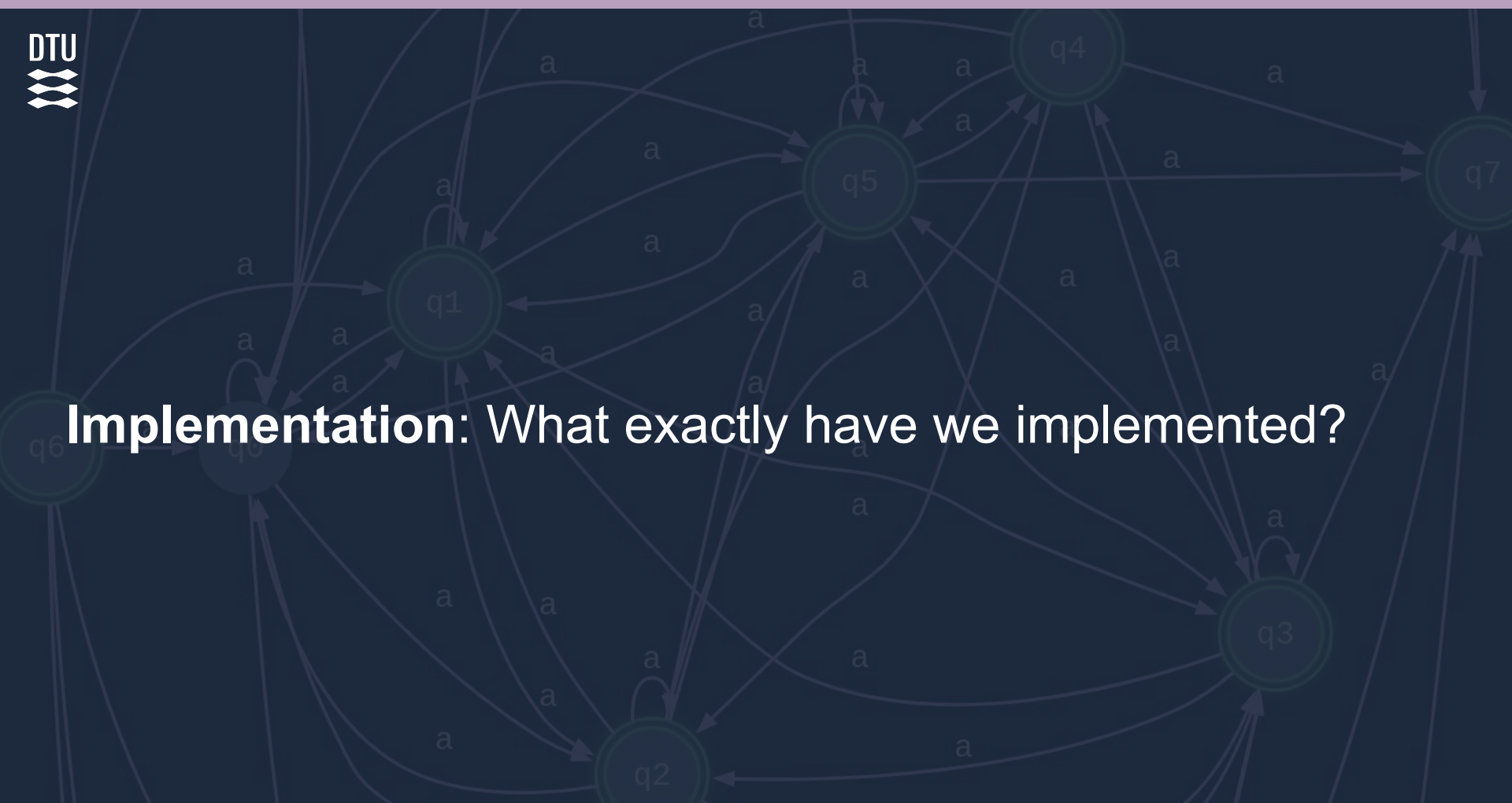
- Do not directly cover our full course-specific pipeline (JFlap, Automata Tutor, FSM Builder)
- Focus on problem instances rather than general implementations (all found)
- Aims at advanced features rather than simple introduction (Spot)
- Some interfaces are outdated or unintuitive for beginners (JFlap, parts of Automata Tutor)

Checkr: Our contributions

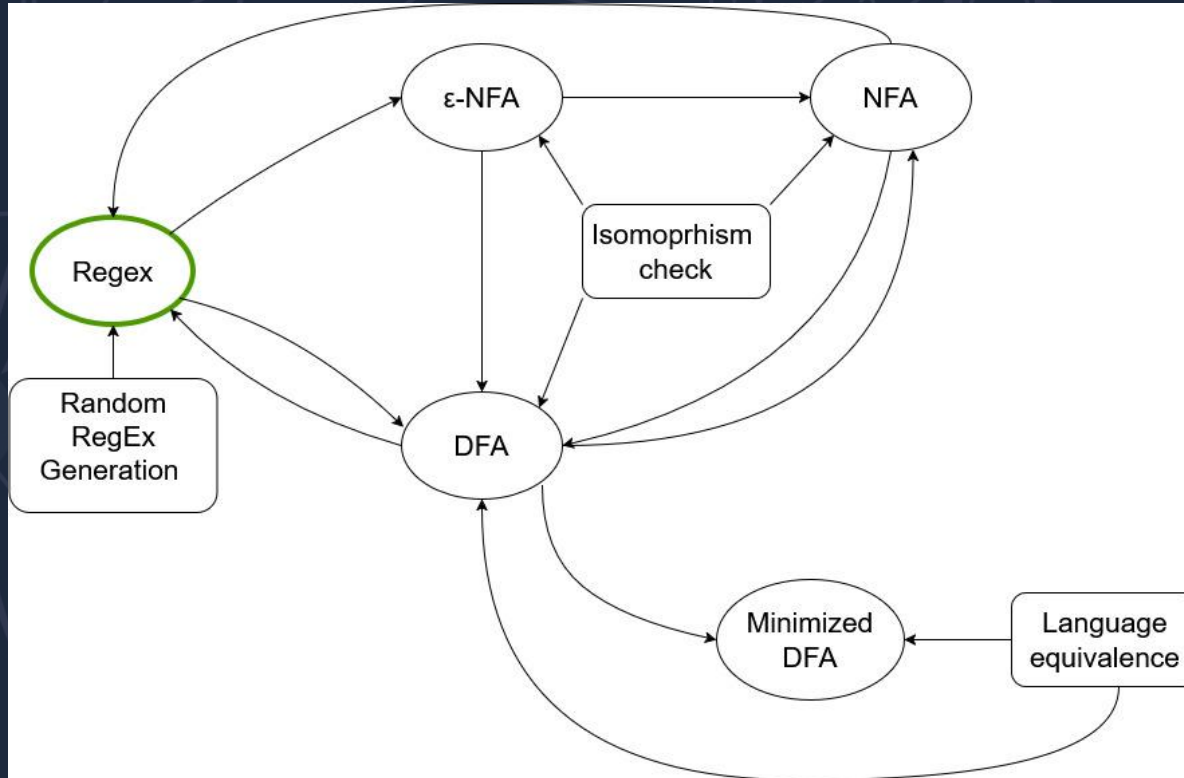




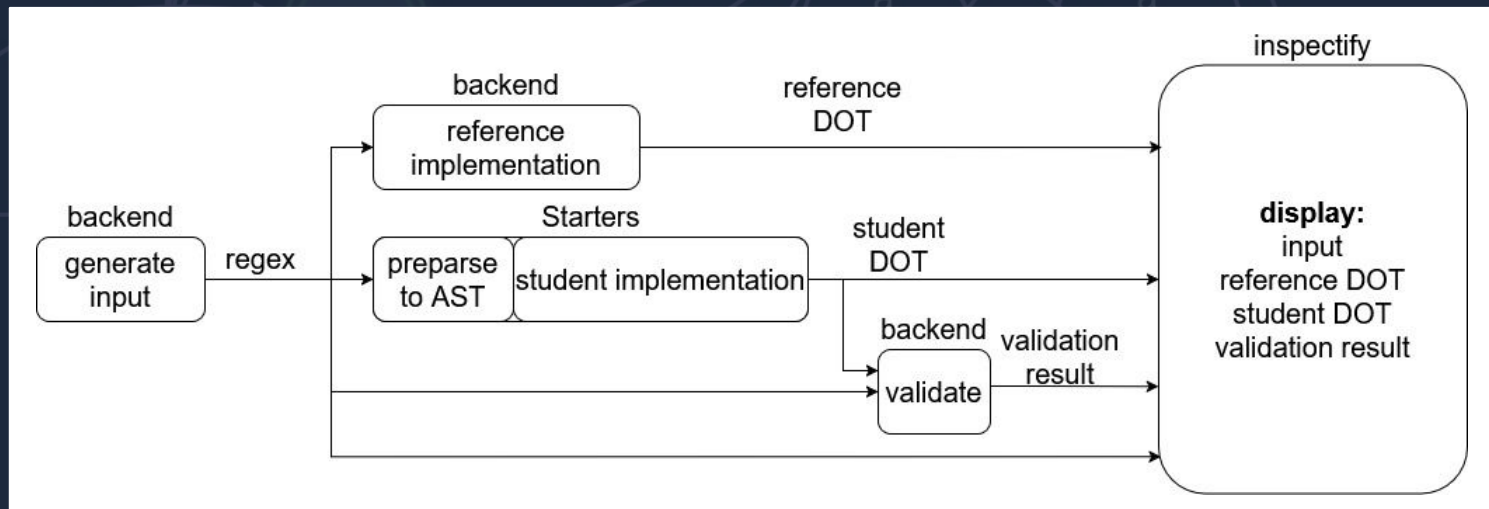
Inspectify: What is it? - DEMO



Implementation: What exactly have we implemented?



Implementation: Parsing



Implementation: Input

- Reuse regex as input of every task
 - Task implementation assumes previous tasks

Allowed characters:

0x20 (space) through 0x7E (~)

Control Characters

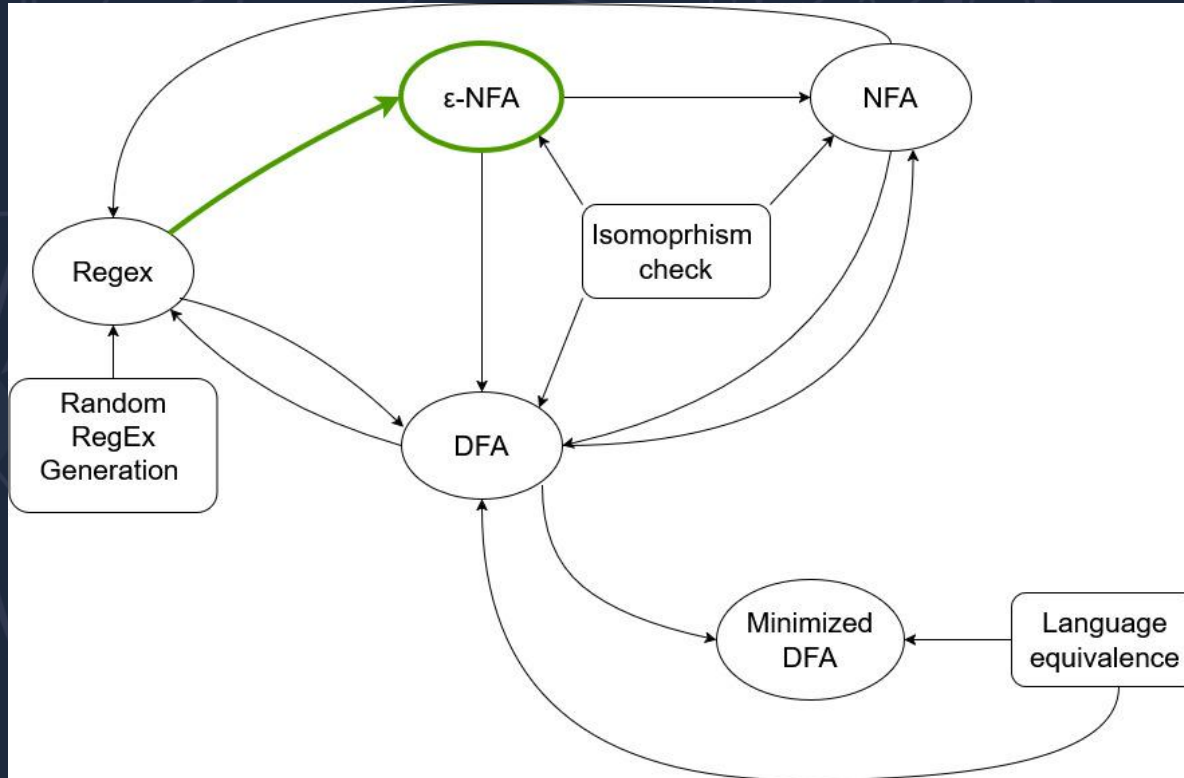
| + * ()

Disallowed characters

" \$. ? [] \ ^ { }

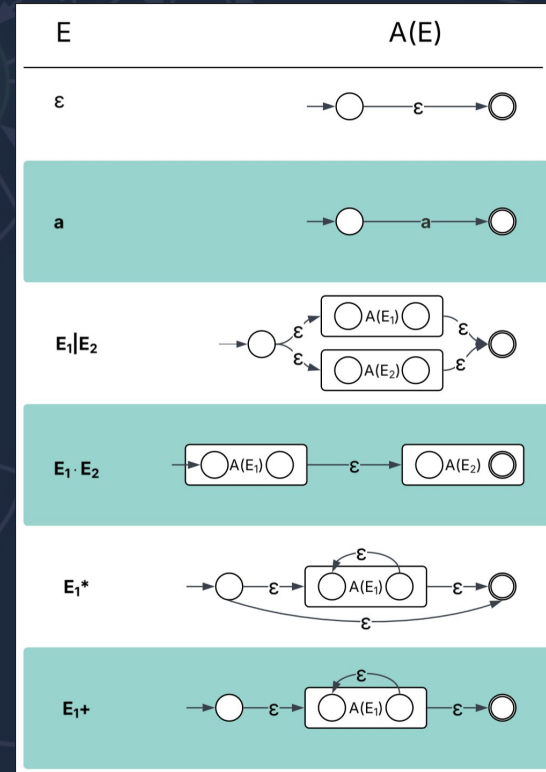
Algorithms: As taught in the course

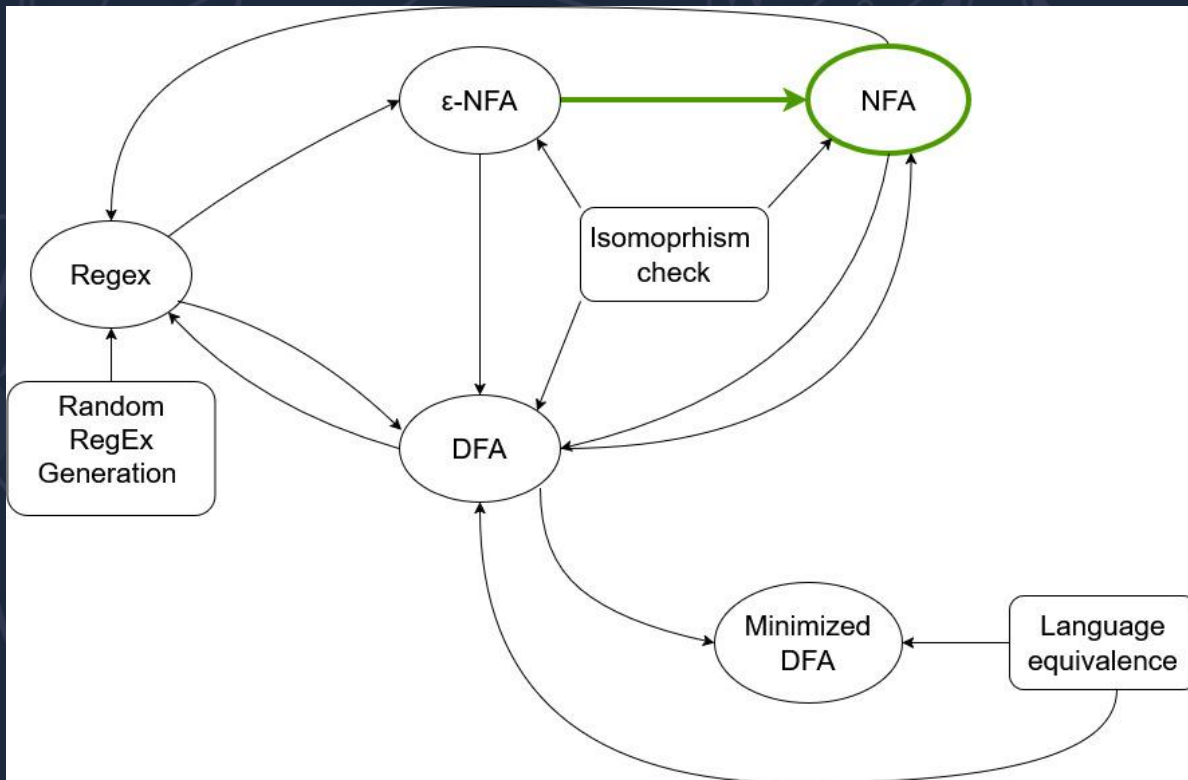
- Matches learning goals
- Easier to maintain
- Potential performance cost negligible



Algorithms: Thompson's

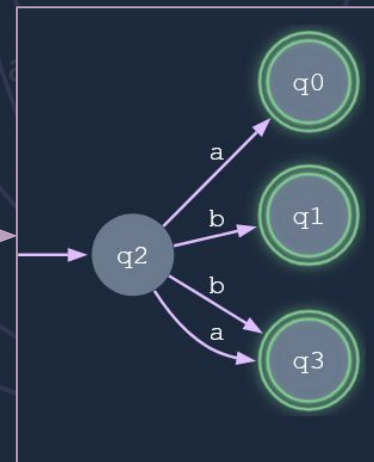
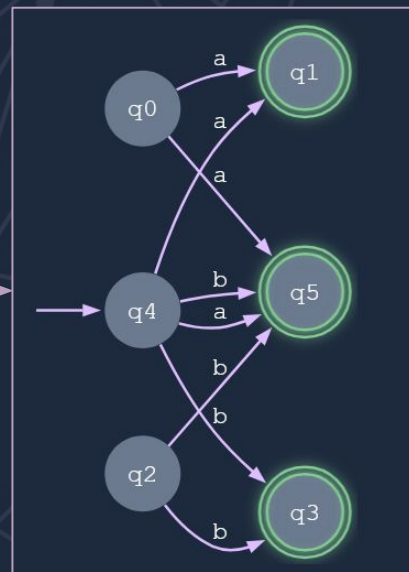
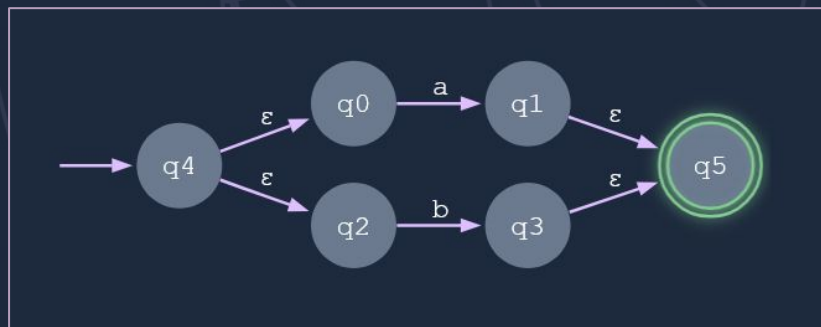
- Why these productions?
 - Matches course material thus aligning learning goals
 - Slight expansion for + to show it has another meaning in common regex syntax

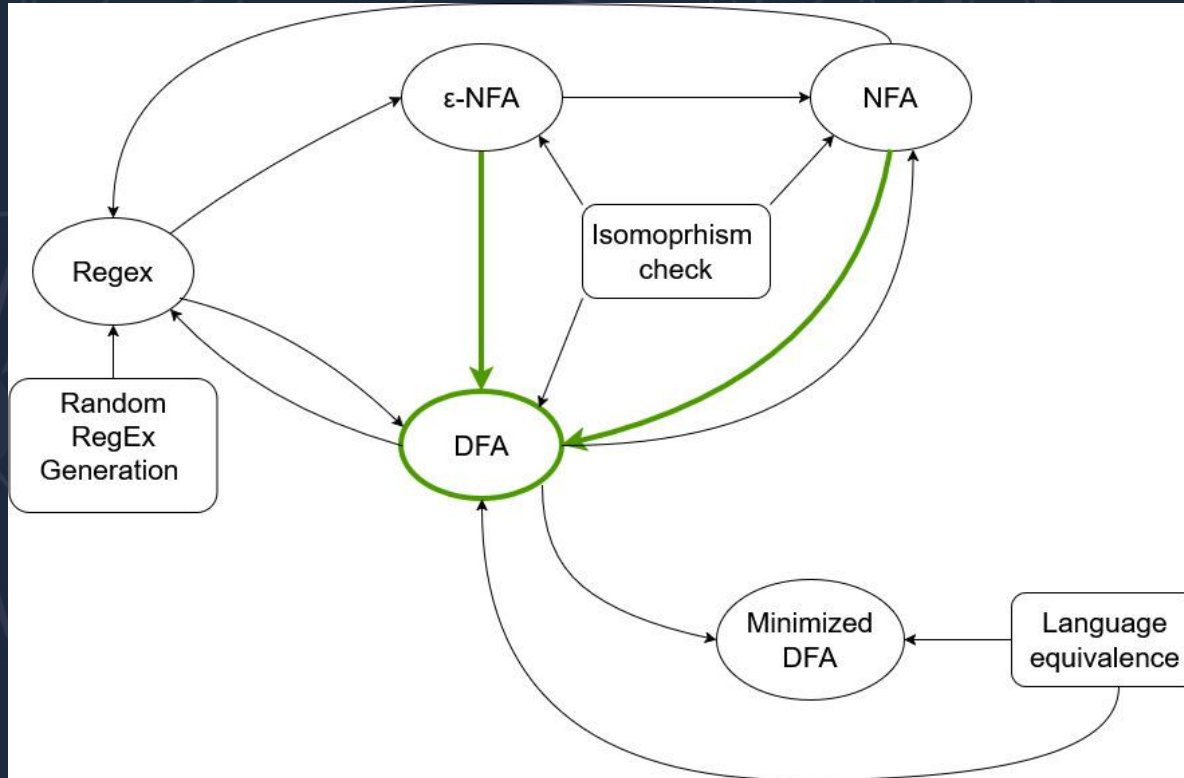




Algorithms: Epsilon removal

- Encode each nodes reachability directly
- Prune unreachable states for readability



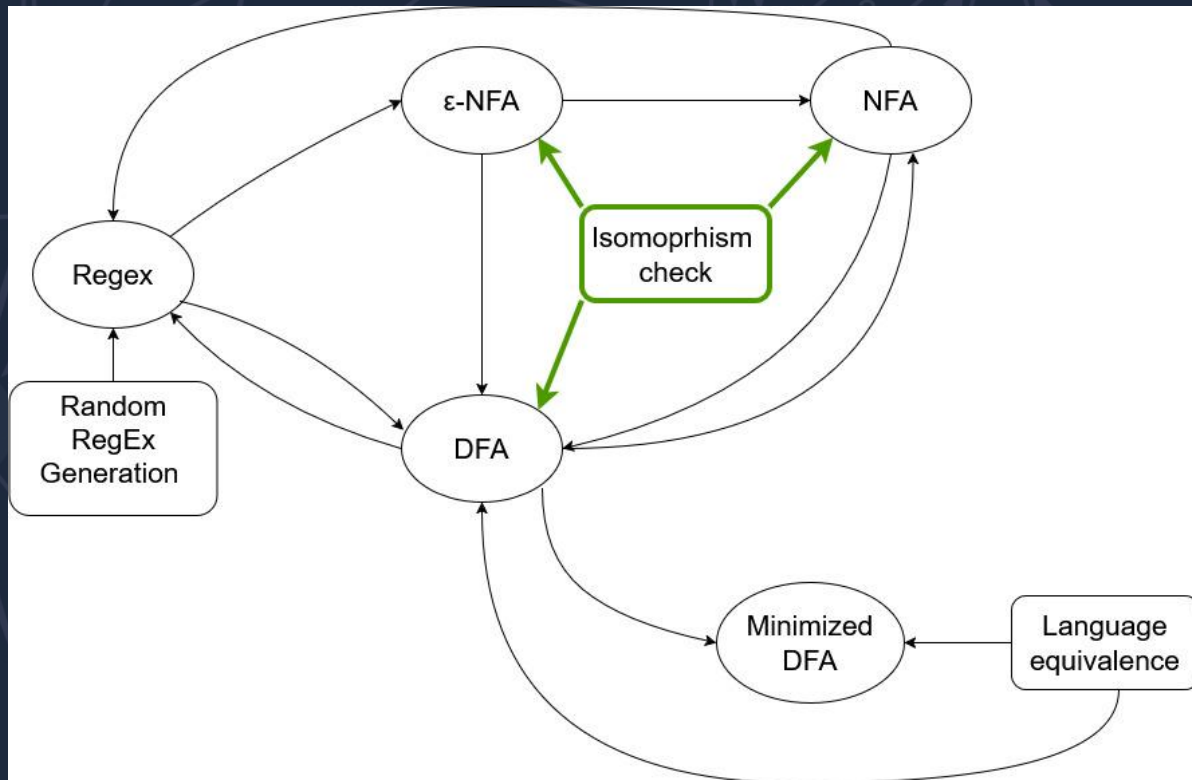


Algorithms: Subset construction

- Include from ϵ -NFA to increase flexibility
- Only compute reachable states rather than all subsets followed by pruning.
 - Better practical runtime



See the algorithms in action: DEMO



Algorithms: Student acceptance through Isomorphism

- Language equivalence
 - Not strict enough (eg. DFA in ENFA task)
- Syntactic equivalence
 - Too strict: Relies on the state set directly
- Isomorphism depends only on structure
 - Structure only, not naming
 - Still strict
 - Must match algorithm
 - Visualization is also structure based

Implementation: Output

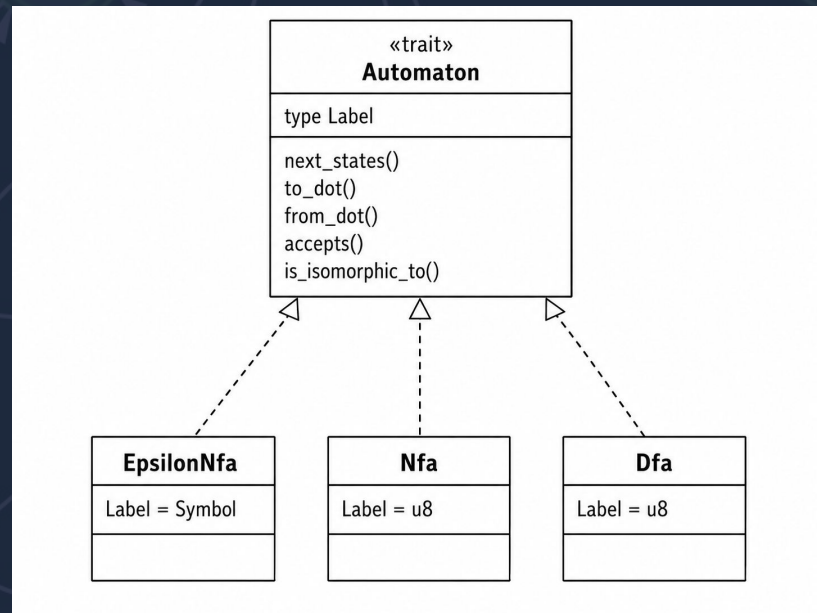
- We choose DOT:
 - Students know it
 - Simple
- We adopt new DOT convention
 - Semantic not visual properties
 - Single label output for simplicity

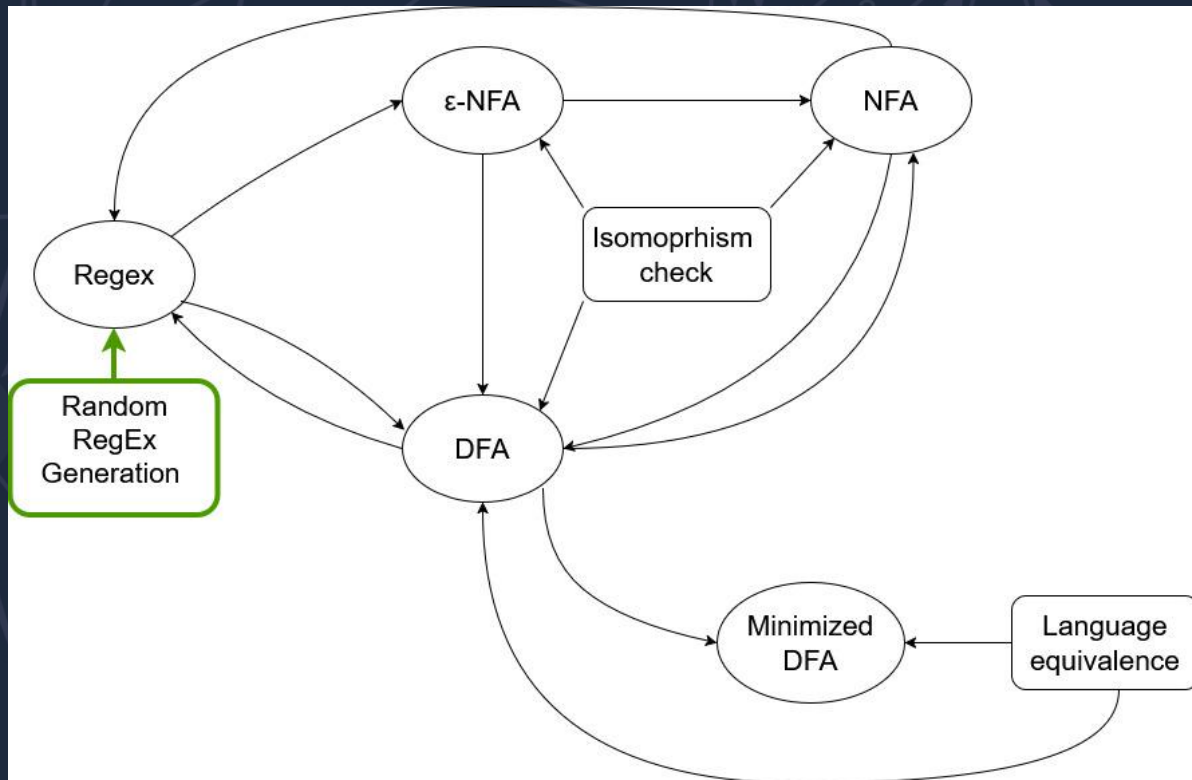
Implementation: Backend

- Data Structure
 - Adjacency List: simple and fast
 - State removal $O(n+m)$ but only done once in pipeline
 - Constant factors weigh more given low graph complexity
- Main transformations
 - Use explicit constructions as used in the course to ease maintenance

Implementation: Backend

- Abstractions
 - Implementers define labels and transition behavior
- Generic DOT handling, word acceptance and isomorphism
- Uses existing dependencies for DOT parsing and isomorphism





Implementation: Input generation

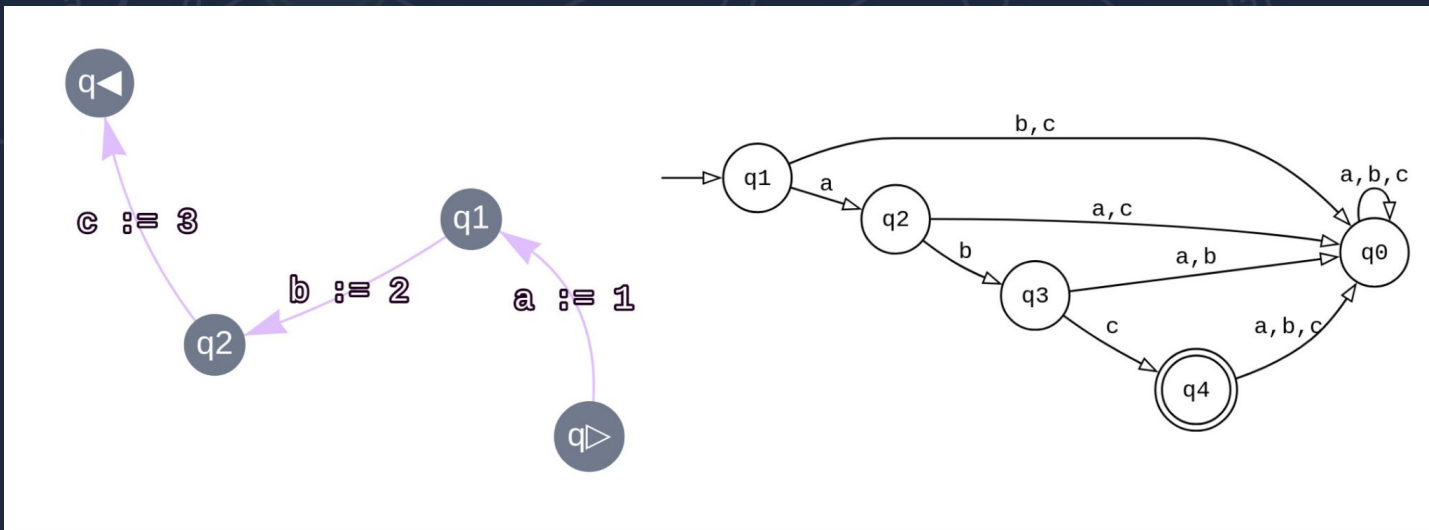
- Generated inputs should
 - Test correct implementation => more complex results
 - Give readable results => less complex results
- Generic and tunable regex generator
 - Result complexity differs by transformation
- Tuning:
 - RegexToENFA: baseline
 - RegexToNFA: Lower recursion depth, as edge count explode
 - RegexToDFA(Direct): More literals to test completeness



Implementation: Visualization DEMO

Implementation: Visualization requirements

- Should resemble automata found in literature
- Should match inspectify theming

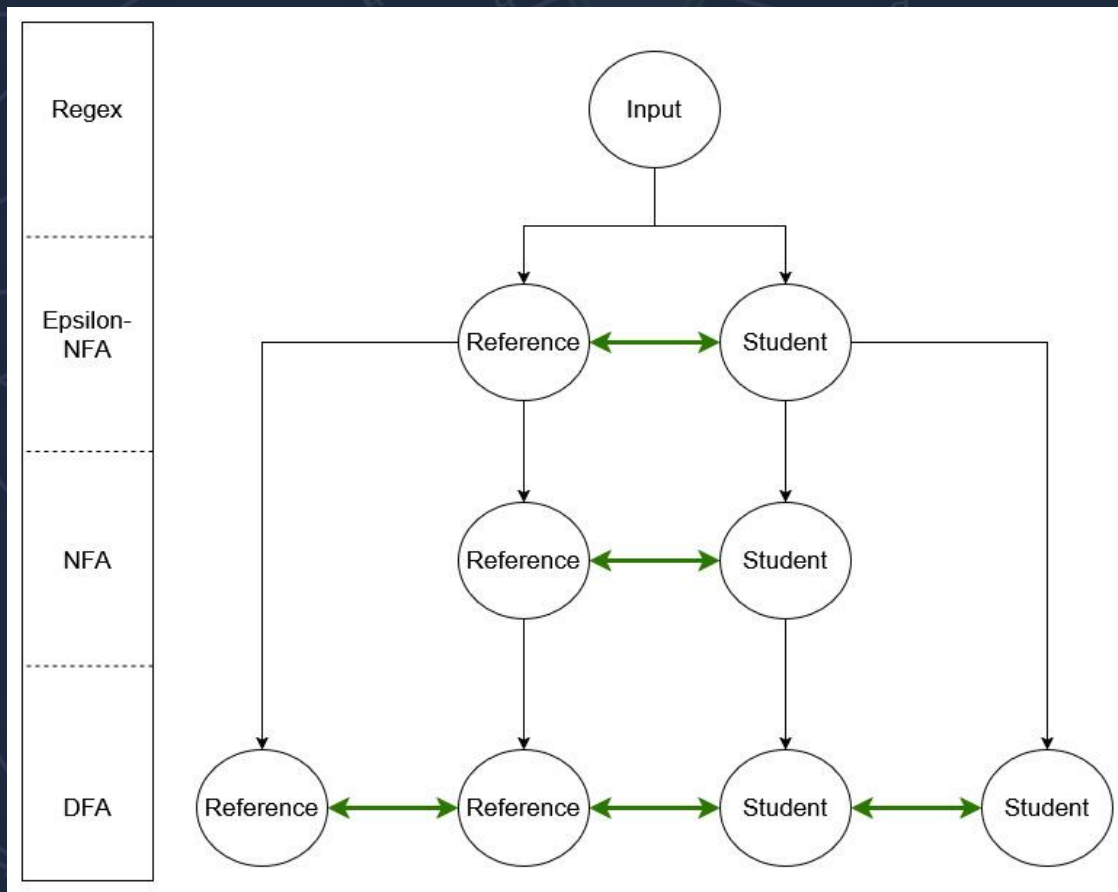




Validation: How can we trust our solution?

Validation: Method is sound

*Green arrow is proven
isomorphism*



Validation: Method is implemented correctly

- Existing:
 - Unit tests, property tests
 - Acceptance agreement tests on random regex and words for all paths and an oracle
 - High code coverage (95%)
- Future:
 - Complete language equivalence test with oracle on random regex
 - Long running property tests

Coverage Report

Created: 2026-05-31 13:10

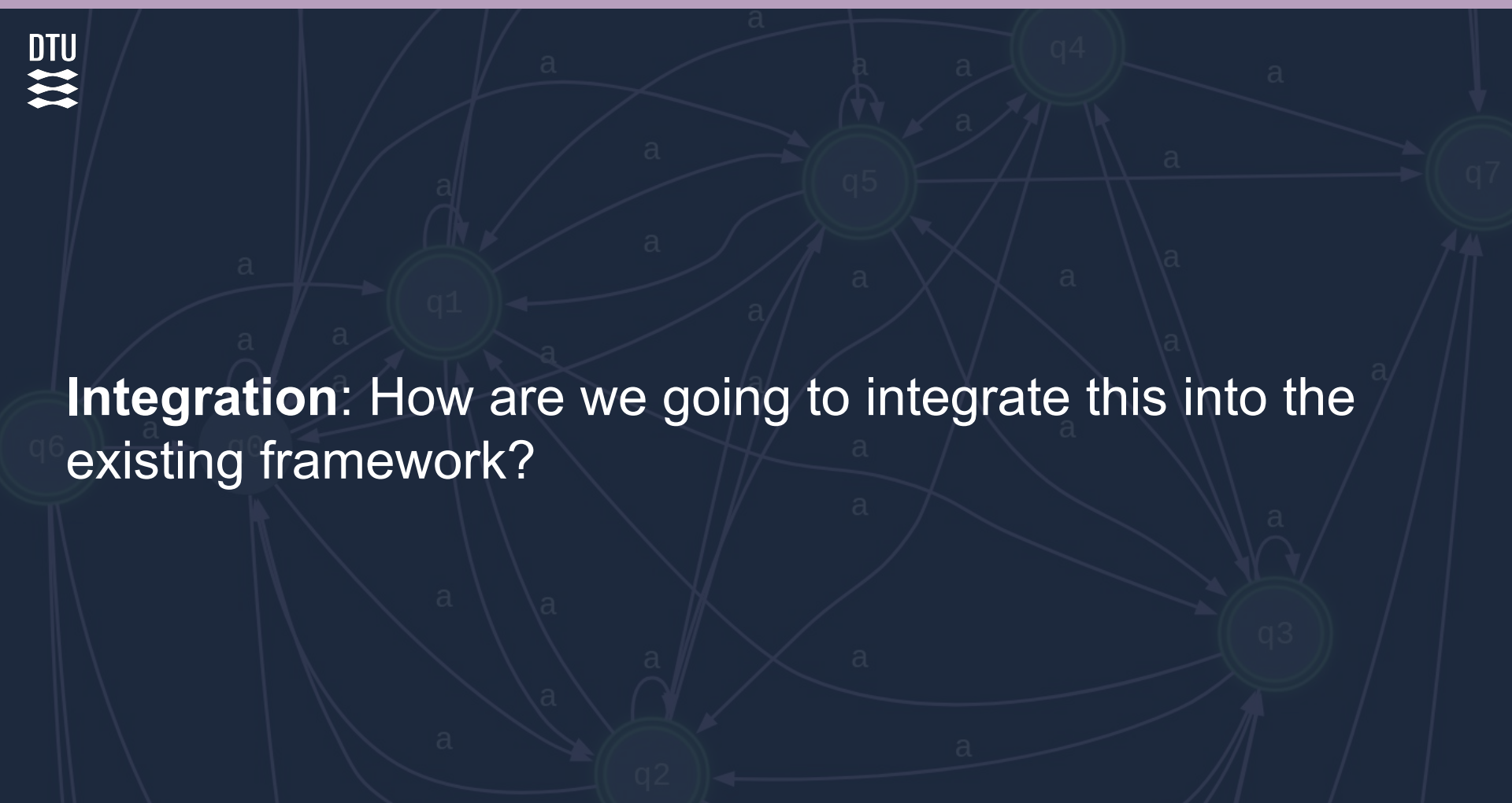
Click [here](#) for information about interpreting this report.

Filename	Function Coverage	Line Coverage	Region Coverage	Branch Coverage
automaton.rs	94.44% (17/18)	97.37% (111/114)	98.63% (216/219)	- (0/0)
dfa.rs	100.00% (14/14)	97.47% (77/79)	97.39% (112/115)	- (0/0)
dot.rs	80.00% (8/10)	88.00% (110/125)	91.53% (216/236)	- (0/0)
enfa_to_dfa.rs	100.00% (12/12)	100.00% (76/76)	100.00% (145/145)	- (0/0)
epsilon_nfa.rs	100.00% (16/16)	100.00% (105/105)	100.00% (154/154)	- (0/0)
epsilon_removal.rs	100.00% (10/10)	100.00% (52/52)	100.00% (95/95)	- (0/0)
errors.rs	33.33% (1/3)	28.57% (4/14)	24.14% (7/29)	- (0/0)
nfa.rs	100.00% (14/14)	97.67% (84/86)	96.77% (120/124)	- (0/0)
nfa_to_dfa.rs	100.00% (12/12)	98.95% (94/95)	98.31% (174/177)	- (0/0)
regex.rs	100.00% (13/13)	90.64% (155/171)	94.71% (340/359)	- (0/0)
Totals	95.90% (117/122)	94.66% (868/917)	95.52% (1579/1653)	- (0/0)

Generated by llvm-cov -- llvm version 21.1.3-rust-1.92.0-stable

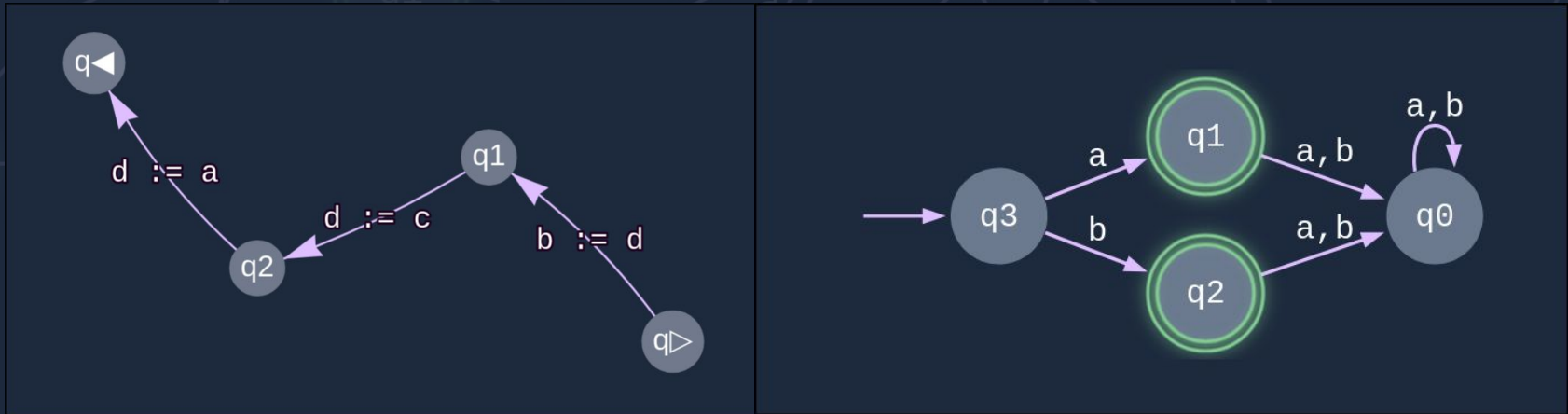


Validation: Visual feedback is meaningful DEMO



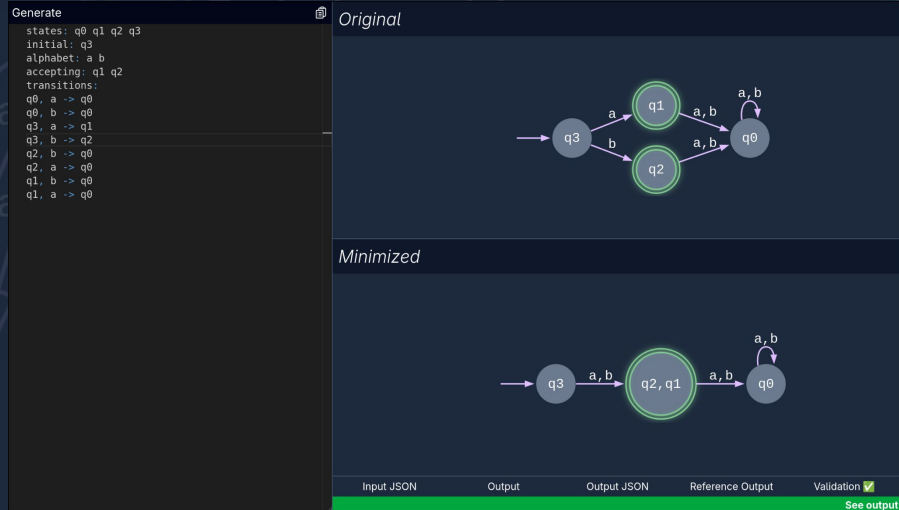
Integration: How are we going to integrate this into the existing framework?

Integration: Matches with inspectify theming



Integration: Integrated with minimizer task

Copy In Automaton Format



Integration: Input evaluation sets

- Input sets made specifically for each environment
- Used to evaluate students

Integration: Student facing descriptions

- Description for input format
- Description for output format
- Descriptions made for all tasks

DOT Language

All automata in this assignment are represented using the [DOT language](#). We use a directed graph format with the following structure:

```
digraph Automata {  
    rankdir=LR;  
    "q0";  
    "q1" [isInitial=true];  
    "q2";  
    "q4" [isAccepting=true];  
  
    "q1" -> "q2" [label="a"];  
    "q2" -> "q4" [label="b"];  
}
```

State Attributes

The following attributes may be applied to nodes:

```
isInitial=true  
isAccepting=true
```

A state with no attribute is an ordinary non-accepting, non-initial state.

Transitions

Transitions are directed edges with a label representing a single symbol:

```
"source" -> "target" [label="label"]
```

For the transitions in ϵ -NFA's you can use the following epsilons: ϵ or ϵ

Notes

- Letting all state names be quoted strings (e.g. "q0") is recommended.
- The graph direction is always left-to-right (rankdir=LR).
- Multiple edges between the same pair of states are allowed (one per symbol).

Integration: Remainders

- All upstream changes are integrated
- Starters already made
- Input generation is tuned for each environment
- References course book directly

Deployment:

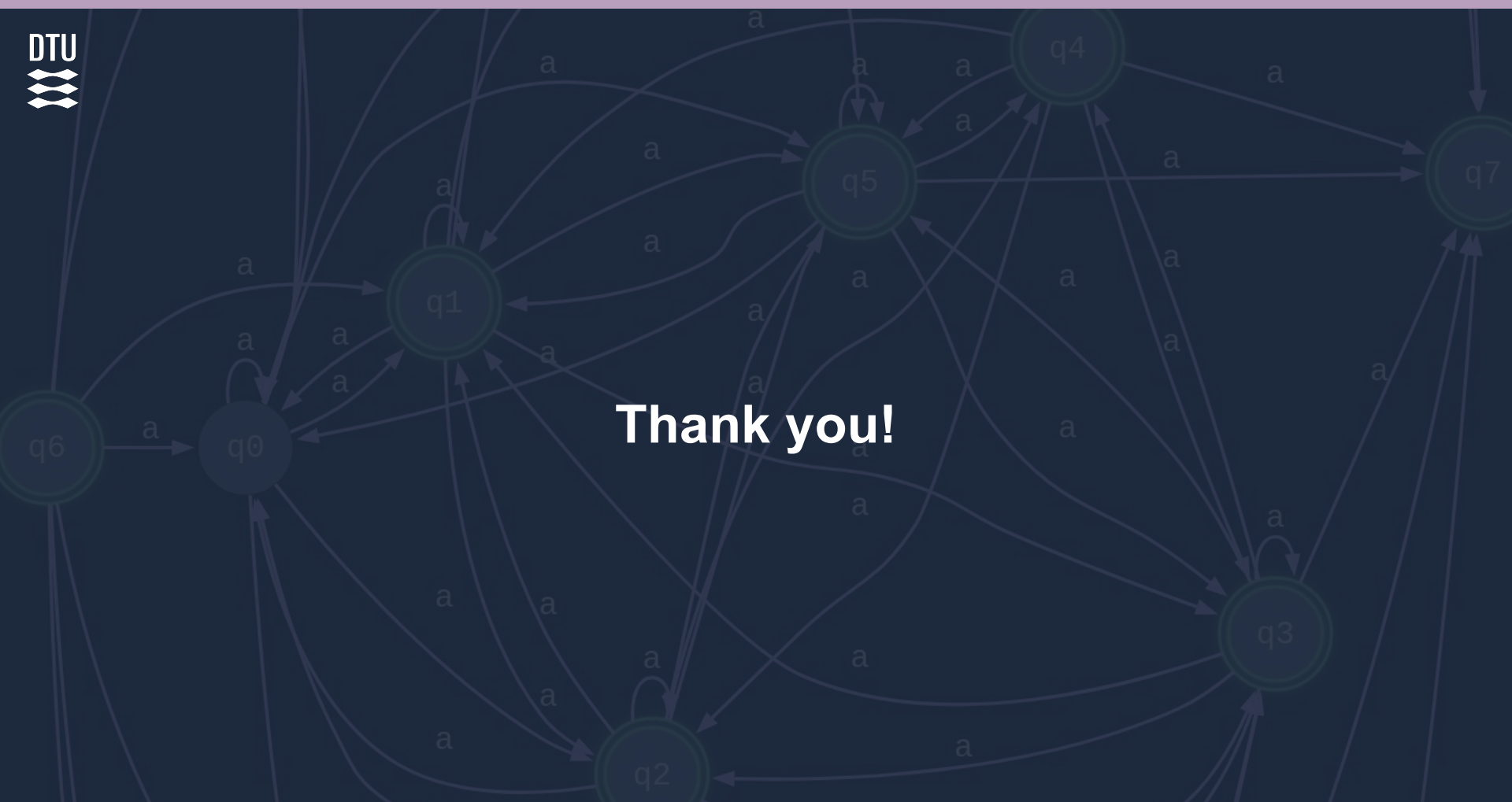
Deployment should be simple

- Some links and task numbers needs to be configured
- Code should be merged
- Some files should be moved

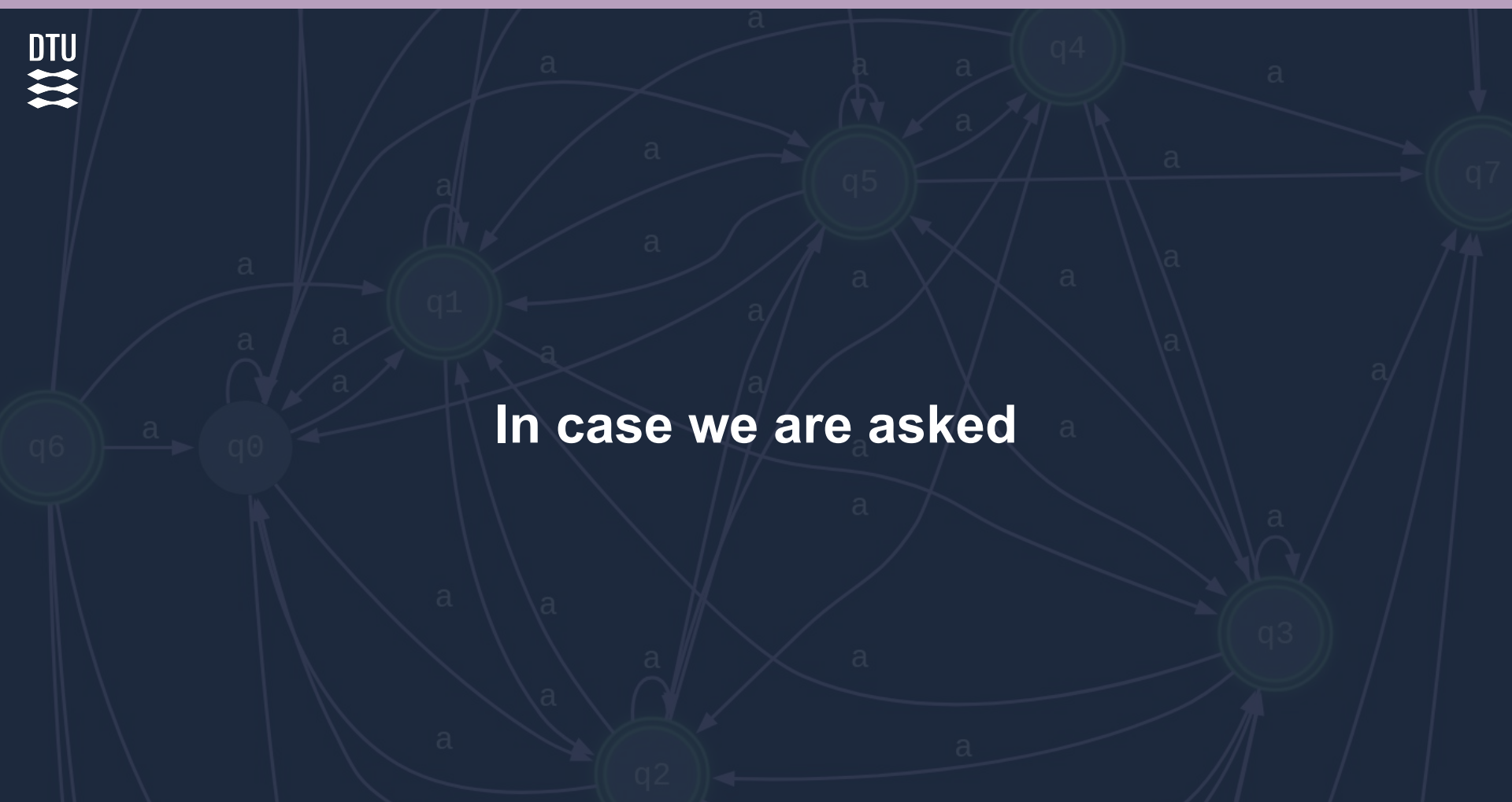
With that we expect that the normal deployment pipeline can run.

Conclusion

- We have implemented key algorithms in inspectify
- Acceptance, input generation and visualization align with learning goals
- Deeply integrated with existing course, checkr and concurrently developed task
- Student evaluation sets and task descriptions made
- Ready to use for teaching!



Thank you!



In case we are asked

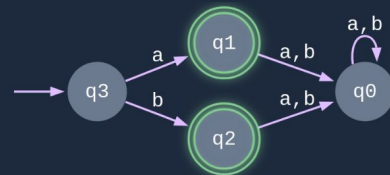
Last isomorphism proof sketch

- Direct route starts in ε -closure($\{q_0\}$); long route starts in $\{q_0\}$.
- The first transitions match because ε -removal encodes ε -closure before and after symbol transitions.
- Afterwards both routes operate on the same ε -closed subset states, so all later transitions match.
- Thompson automata have no incoming transitions to the initial state q_0 .
- Since q_0 cannot be reached again, difference is isolated to the initial DFA state.

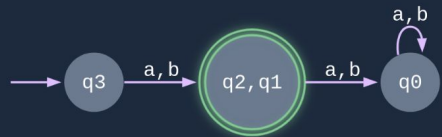
Preserving state names

- Makes pipeline chosive
- Less clear reference comparison?

Original



Minimized



Input JSON

Output

Output JSON

Reference Output

Validation ☒

[See output](#)

Expanding productions

- More general repetitions
 - eg. $r\{5,10\}$
 - Table shows $r\{n,m\}$
- Classes?
 - eg. $[a-z]$, $[0-9]$
- Wildcard?
 - Interaction with alphabet?

Case	Expansion	Meaning
$n = 0, m = 0$	ε	No repetitions left
$n > 0$	$r \cdot r^{\{n-1, m-1\}}$	Add one mandatory copy
$n = 0, m > 0$	$(r \mid \varepsilon) \cdot r^{\{0, m-1\}}$	Add one optional copy

Any expansion requires re-tuning input generation

Future work: Partial Credits

- Give partial credit for language equivalence
- Requires general chekr change
- Inspired by FSM builder that used language similarity for partial credits

Future work: Structure dependent visualization

- Show canonical form
 - Computationally hard
 - We have an idea of how to do it
- Not actually meaningful
 - Feedback when structure is different would be meaningful
- Structure similarity scheme
 - Could be meaningful
 - We are not quite sure how

HOW: Structure dependent visualization

- Idea in report: make sorting key with enough structural information to make pseudocanonical form
- Limited to predetermined local information.
- To find canonical form, chose the ordering whose serialization of structural representation is lexicographically smallest.
 - But there are $n!$ permutations of n numbers
- Combine the two approaches:
 - Limit search space by using structural sorting keys
 - choose canonical form in reduced search space
- Likely no good theoretical guarantees, but decent practical performance seems likely

Runtimes

- Thompsons: $O(\#AST \text{ nodes})$
- Epsilon removal: $O(nm(n+m)) = O(n^3)$,
 - Can be: $O(n(m+n)) = O(n^2)$
 - extra m from recalculating closure after taking symbol transitions
- Subset Construction: $O(2^{cn})$
- Isomorphism: VF2: $O(n!n)$, $\Omega(n^2)$
 - First check that state counts agree
- Edge compaction: $O(m)$ Expected
- Dead state removal: $O(nm|F|)$