Parallelizing NFA to DFA Conversions

Alexis Gadonniex, Nikhil Mehta

December 22, 2022

1 Problem Statement

We seek to parallelize the subset construction algorithm that is applied to a nondeterministic finite automaton (NFA) to convert it into a deterministic finite automaton (DFA). This is particularly useful in the context of string processing and compilers where one wants to check whether a given string matches a regular expression or not. Since regexes can be represented as NFAs, this is the same as asking the question of whether a given string will be accepted by an NFA. At the surface this might seem like a tricky problem since NFA's are, as their name suggests, non-deterministic. However, NFA's can always be converted into DFAs through the subset construction algorithm. This algorithm, described below, is one that has to consider a variety of possible intermediary states in the NFA at the same time. Thus, the algorithm is ripe for parallelization. To generate NFAs in the first place we will use three techniques. First, we will implement non-parallelized functionality to convert a regex to an NFA using Thompson's algorithm. Second, we will create another type of NFA directly from a dictionary of all accepted strings in a language. Third, we will directly generate NFA's with random edges and transitions.

Thus our project requires several parts. First, we need to create a data structure to represent automata whether they are deterministic or not. Then, we need write single-threaded algorithms to generate NFAs. At the core of our project, we need to write a single-threaded version of the subset construction algorithm and then work to parallelize it. Last, we opted to write an algorithm to check if strings are accepted by a given automaton.

2 Algorithm

2.1 Subset construction

The subset construction algorithm is used to convert an NFA into a DFA. The subset construction algorithm works by considering sets of states. That is to say, for a given NFA consider the set of all states reached from the starting state. That constitutes the starting state for your DFA. Then consider all the states (if any) accessible from the initial set of states that are reached when you pass in a given character. That constitutes the next state of the DFA. Then, consider the set of all states reached when you pass in a different character. The algorithm continues to reapply this step to find all possible sets of states in the NFA for all possible sequences of characters. Each set of NFA states then corresponds to a unique state in the DFA.

Below we present pseudocode of the algorithm:

```
Init: Add epsilon-closure of the start states to DStates

While there is a non-visited state T in DStates do
  mark T as visited
    for each symbol c do
      compute S the set of states that can be reached from T through c
      compute SE the epsilon-closure of S
      if SE is not in DStates do
            add SE to DStates as a non-visited state
        mark SE as final if it contains a final state of the NFA
```

```
      end
      Transition[T, c] = SE
    end
  end
```

The algorithm has a worst-case complexity of O(2^n) since the number of subsets of an n-state NFA is 2^n. This worst-case complexity can be achieved with fairly simple languages and this will be useful to test our code against difficult instances.

2.2 Thompson's rules

While not a parallelized feature of our code, we use Thompson's algorithm to generate NFA's from regexes. Thompson's algorithm is a set of conversions that can be applied to each regex rule to generate an NFA. Thompson's algorithm can be applied recursively to a syntax tree of a regular expression. Below we describe the four rules for Thompson's algorithm.

To represent two sub-regexes $N(s)$ and $N(t)$ that are concatenated with one another one can use the rule shown in Fig. 1. To represent two sub-regexes $N(s)$ and $N(t)$ that are joined via a union, sometimes referred to as "or" and typically denoted by | , one can use the rule shown in Fig. 2. To represent a sub regex $N(s)$ that is encased in a Kleene Enclosure, normally denoted by a *, one can use the rule shown in Fig. 3. Finally, to represent the transition of an accepted symbol a, one can apply Thompson's symbol rule shown in Fig. 4

We further discuss Thompson's algorithm and how we implemented it in section 4.

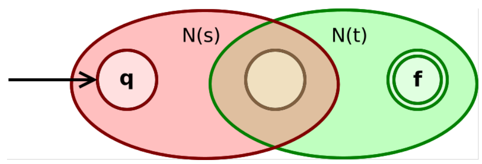

Figure 1: Thompson's concatenation rule

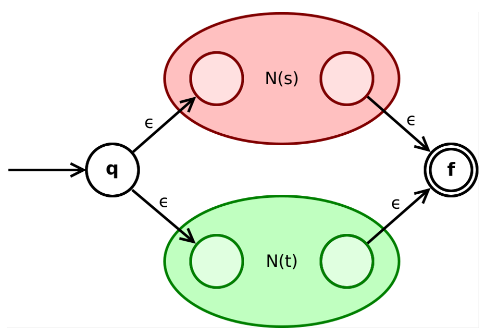

Figure 2: Thompson's union rule

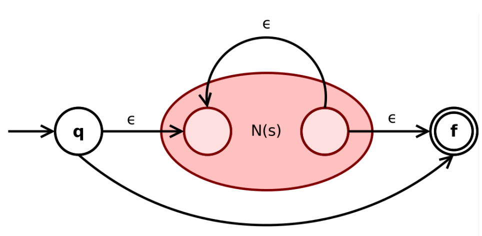

Figure 3: Thompson's symbol Enclosure rule

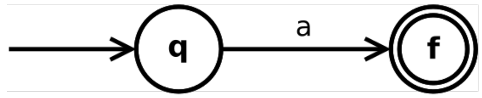

Figure 4: Thompson's Kleene Enclosure rule

3 Implementation

3.1 Automaton Data Structure

To represent NFAs and DFAs in our project, we wrote an Automaton data type in Haskell in `Automaton.hs`. `State`s in an `Automaton` are represented as unique integers. A transition is denoted as an `Edge` which consists of a tuple of a `Label` and the next `State`. A `Label` is either an `Epsilon` or a `Label Int` where the integer denotes the character in the alphabet that is labeling a given transition. An `Automaton` consists of an `Adjacency List` which is a Map with each `State` serving as the key and their subsequent list of `Edge`s serving as the value. An `Automaton` also takes a list of `Label`s to represent the alphabet. Last, an `Automaton` takes a Set of final states and a Set of start states.

3.2 Subset Construction

From the very beginning, we tried to implement the subset construction algorithm in a way that could be easily parallelized in the future. Thus, we tried to isolate some `map` patterns.

We used a layer-by-layer approach, similar to a BFS, that we divided into two phases:

- An exploration phase, where we explore all the transitions leaving from the previous layer. We create the cartesian product between the alphabet and the DFA states of the previous layer, and for each pair, we compute the target DFA State (set of NFA States) by performing look-ups to the NFA's transition table and ε-closure. We parallelized this phase by running these computations using `parMap` and the `rdeepseq` strategy.
- A Sequential phase where we will try to merge all the newly discovered states from the parallel phase with our partially created DFA. In order to keep track of the mapping between a set of NFA's states (set of integers) and the corresponding DFA state (an integer), we use a Haskell `Map` with `Set`s of `Int`s as keys and `Int`s as values. A newly discovered DFA state can lead to several results: it is a new state and should be used in the next exploration phase; it is empty so we can ignore it; it already exists elsewhere in the DFA (a previous layer or the same one) and we just have to add the edge to our adjacency list.

We also tried to explore several layers at once during the exploration phase to create more sparks but it was not very efficient because the number of states in each layer is growing very quickly and most of them are actually redundant. It is more effective to sort the new states from the already existing one at each layer. We illustrate our algorithm working on a sample NFA in Fig. 5

A clear bottleneck in our implementation is the sequential phase which limits how much we can get from parallelism (Amdahl's law). The most expensive operation during this phase is the look-up in the "NFA state sets to DFA state", especially if the DFA States are made of a large number of NFA states. A possible improvement would be to use some kind of hash function for a set of integers to produce smaller keys for the Map. This hash could be computed during the parallel phase. Unfortunately, we didn't have time to explore this possibility.

Reducing the time spent in this phase was one of our main concerns when considering the various options we had to generate NFAs.

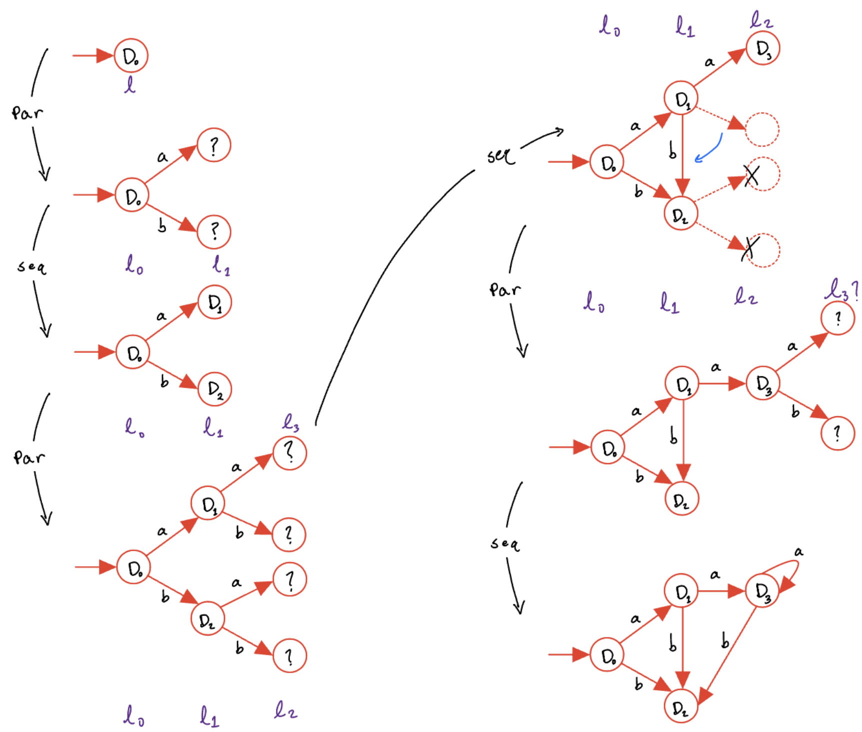

Figure 5: Subset construction implementation diagram

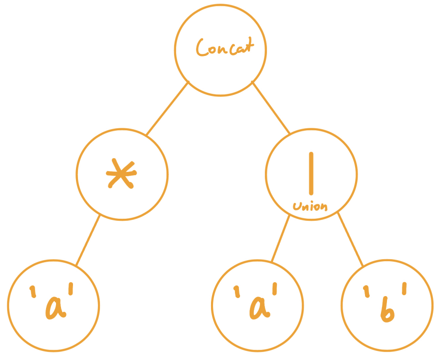

Figure 6: Syntax tree generated from "a*(a|b)"

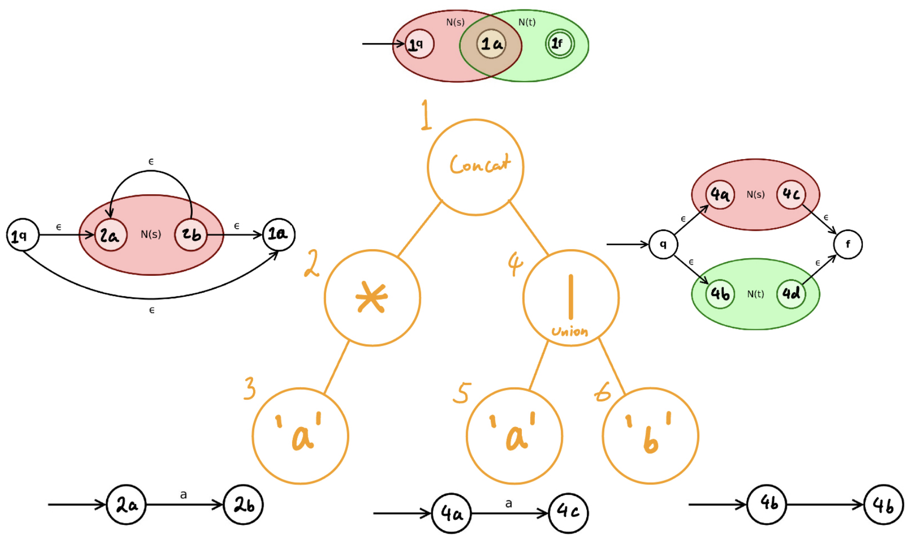

Figure 7: Traversal of syntax tree from Fig. 6 with corresponding Thompson's construction at each node.

4 NFA Generation

4.1 From regular expressions

We used three techniques for generating NFAs. Our first method involved using Thompson's algorithm to generate an NFA from a regex. To achieve this we wrote a parser `regexParser` in `Thompson.hs` using the Haskell `Parsec` library to take in a regex as a string and output a syntax tree. We used the standard notation of * for kleene enclosure and | for union. Our parser supports the use of parentheses to enforce a certain precedence. Characters of the accepted alphabet for the regex could be denoted as themselves. Concatenation is implicit when elements of the regex succeed each other. An example regex could be: `"a*(a|b)"`. We show the resultant syntax tree for this regex in Fig. 6.

The resultant syntax tree is then fed into our implementation of Thompson's Algorithm in `thompsons` which is called by `makeThompsonNFA` in `Thompson.hs` which traverses the syntax tree using a depth first search and builds up the adjacency table for NFA recursively. To generate the alphabet for the NFA we used a separate DFS to search for all characters, a.k.a. leaf nodes on the syntax tree. We show an example DFS with the each node in the tree from Fig. 6 labeled according to the order it was visited in Fig. 7 We then show the resultant Thompson's construction for that node.

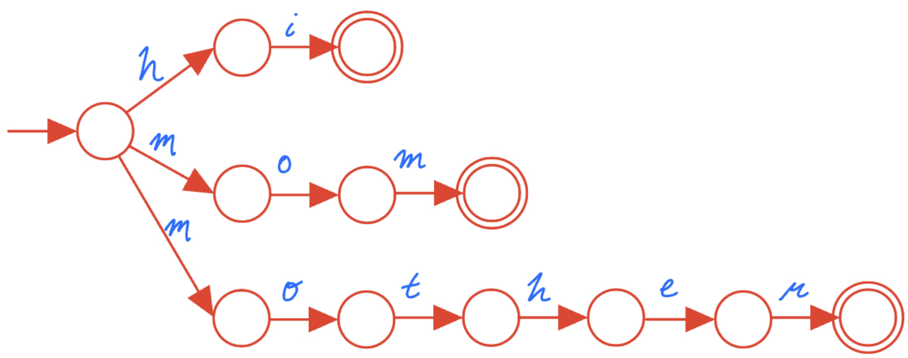

Figure 8: Sample NFA created from a dictionary of all accepted words in a language.

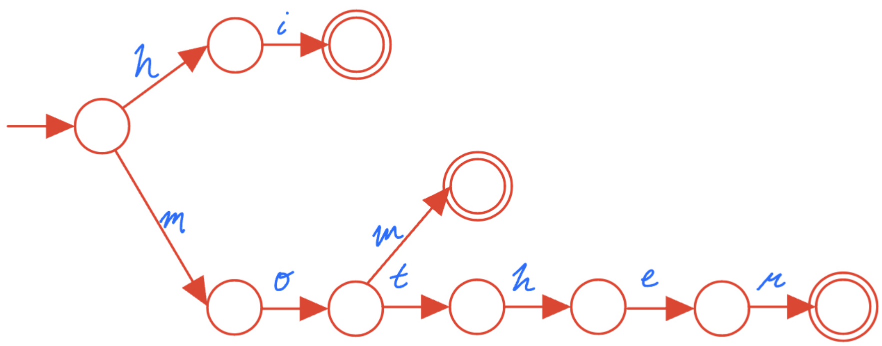

Figure 9: DFA converted from NFA above from dictionary of all accepted words in a language.

We quickly realized that, in order to create sufficiently large problems for our algorithm, we would need to create random regular expressions. One way of doing that could be to chose a number of terminal states and, in a recursive fashion, randomly split the remaining nodes between the two branches of an operator (concatenation or `|`). However, we were afraid that the structure of this type of NFA would be too linear for our algorithm to produce interesting results. We chose to spend time on other methods.

4.2 From a list of words

Our second method of generating NFAs was to build it directly from a dictionary of all the accepted strings in the language. We implemented the function `makeDictNFA` in `Dictionary.hs` which takes in a `[Char]` of all the strings from a dictionary. Then, each string is added onto the NFA from the starting state via an epsilon transition. For example, the following input to `["hello", "world", "mom"]` would generate the NFA shown in Fig. 8.

This is a pretty straightforward way of generating NFAs with a very regular structure. The results are discussed below.

4.3 From a given size and density

The last method we tried was to generate a fully random NFA, without any particular meaning, from a set of parameters. The parameters we chose to expose are:

- The number of states

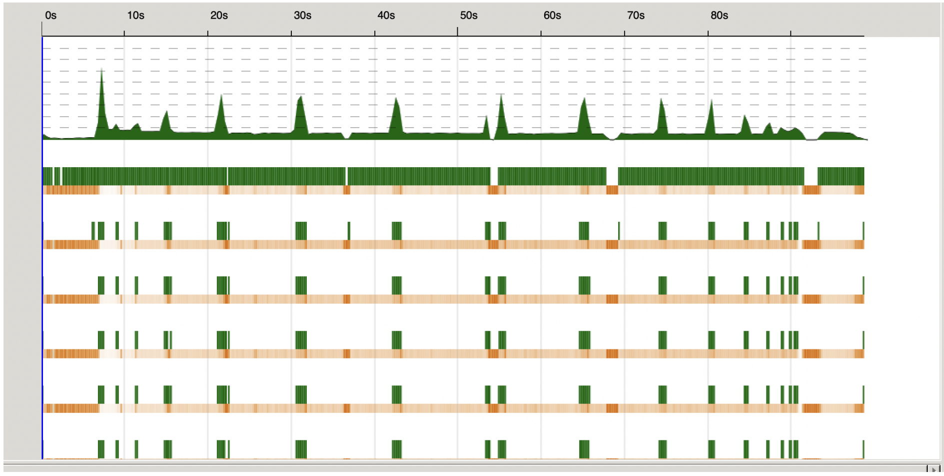

Figure 10: Most time spent in sequential phase when using an NFA created from a list of words

- The size of the alphabet
- The probability that, given two states `s1` and `s2` and an element of the alphabet `c`, there is an edge from `s1` to `s2` with a label `c`.
- (The number of initial states)
- (The number of final states)

The last two are not very important since they don't have any noticeable influence on the complexity of the algorithm.

Since our implementation is a bit naive, we have to use high probability values (it depends on the size of the alphabet) in order to keep our graph connected and avoid isolated states in the NFA.

5 Results

5.1 List of words

The NFAs built from lists of words looked like a perfect use-case for our algorithm. But it turned out that the result were a lot worse than we initially expected. You can see in the threadscope graph in Fig. 10 that the algorithm was running on a single core most of the time. The main reason why it spends all this time in the sequential phase is that each DFA state is made of a lot of NFA states. To actually see the effect of parallelism, we had to consider very long lists of words (200000 words). But the problem is that it leads to huge NFA States (for example: about 10000 words start with the letter `a`). In this case, the look-ups are extremely expensive and the memory usage is extremely high (maximum heap size 3Gb; total allocated 200Gb !!). We can also note that there are 20 million sparks created, half were converted and the other half overflowed. We tried using `IntSet` as keys but it didn't significantly improve the results.

5.2 Random NFA

When using the random NFAs we described above, we were able to tune more precisely the parameters to find NFAs that fit our algorithm.

The best results were obtained for NFAs with 600 states, an alphabet of size 20 and a probability of 0.4. We obtained a speed-up of 3 using 8 cores (on an M1 Macbook Pro with 8Gb of RAM). You can see on Fig. 11 that most of the time is spent in the parallel phase. The speed-up for different number of threads is plotted on Fig. 12. Note that we could get a slightly better speed-up by generating the random NFAs in advance and parsing them from a text file since it takes about 10 percent

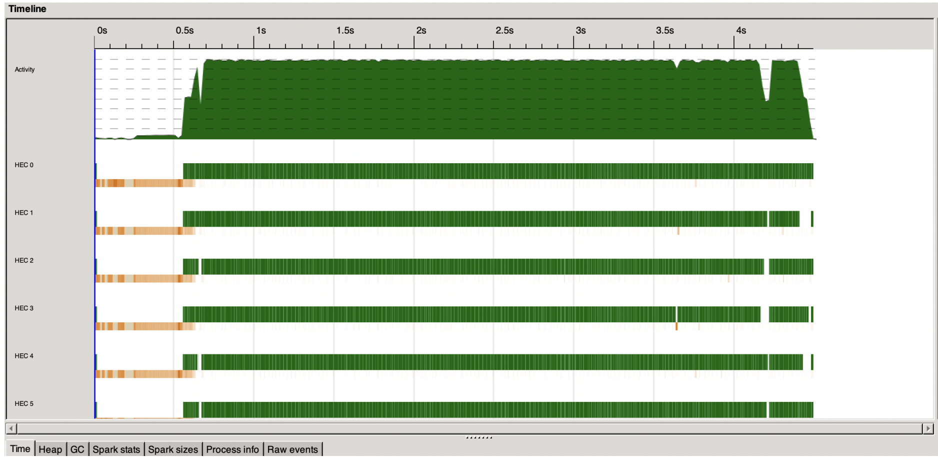

Figure 11: Threadscope on 8 threads for a random NFA with 600 states, alphabet of size 20 and probability of 0.4

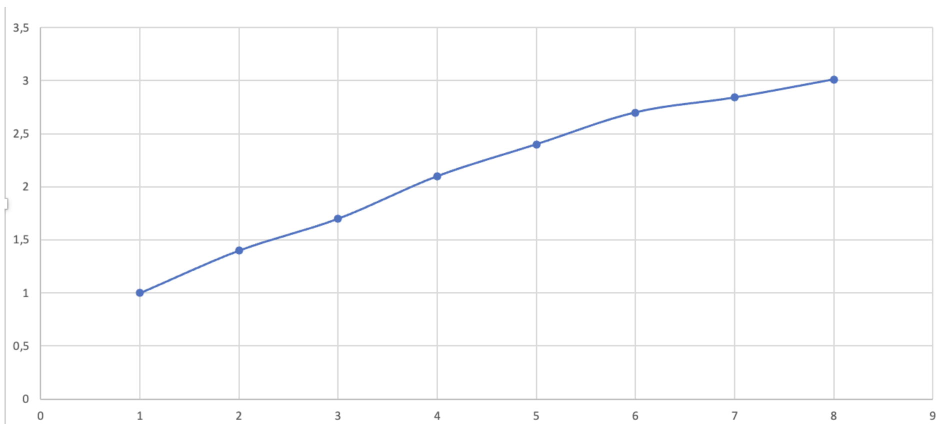

Figure 12: Speed-ups from 1 to 8 threads. Measured on a sample of 10 random NFAs with 600 states, alphabet of size 20 and probability of 0.4

of the total time and is fully sequential.

These are relatively small NFAs but highly interconnected. This leads to pretty small DFA states (in term of number of NFA states) but a lot of different combinations of those. The average size of a state is about 60 NFA states, a lot smaller than the NFAs from a list of words (several hundreds in average and up to several thousands for about the same computation time).

Changing the number of states of the NFA has a relatively small impact on the results. Below 500 the algorithm finishes too quickly for the parallelism to make a real difference. Above (we tested up to 2000 states), the computation time increases very quickly but the speed-up stays the same. It takes about 10 times longer when going from 500 to 1000 states.

We tried to modify the probability but it is a very sensitive parameter and shows the limitations of our modelization. If we decrease the probability too much the graph becomes poorly connected and we end up with a small number NFA states that are truly relevant. In the contrary, if we increase the probability too much, the graph becomes too connected and there are only a small number of DFA states to explore.

The alphabet size is also tricky since increasing it leads to an exponential number of additional edges.

We chose to stick with numbers between 20 and 30 since it is the range of the latin alphabet.

6 Usage

Our code is intended to run with GHC 8.10.7, HLS 1.8.0.0, Stack 2.9.1, and cabal 3.6.2.0.

To build our code run `$ stack build` in the base directory. To run our code using the following command: `$ stack exec subset-construction-exe -- +RTS -ls -s -lf -N8`

7 References

We cite the following sources and thank the respective owners:

1. https://en.wikipedia.org/wiki/Thompson%27s_construction for the Thompson's construction graphics

2. Professor Stephen Edwards PLT slides for sample Regexes used in figures and testing

8 Appendix

/src/Automaton.hs

```haskell
module Automaton (Automaton(..), AdjacencyList, DfaStatesMap, State, Edge, Label(..),
    exampleAutomaton, exampleAutomaton2, ioDumbAutomaton) where
import           Control.DeepSeq (NFData (rnf))
import qualified Data.List       as List
import qualified Data.Map        as Map
import qualified Data.Set        as Set

-- A node's "id"
type State = Int
-- A labeled edge pointing to a node
data Label = Epsilon | Label !Int deriving (Eq, Show, Ord)

instance NFData Label where
  rnf (Label x) = rnf x
  rnf Epsilon   = ()

type Edge = (Label, State)
-- Successors map, initial states and final states
type AdjacencyList = Map.Map State [Edge]
data Automaton = Automaton !AdjacencyList ![Label] !(Set.Set State) !(Set.Set State)
    deriving (Show)

type DfaStatesMap = (Map.Map (Set.Set State) State, Int)

-- Example taken from https://en.wikipedia.org/wiki/Powerset_construction (5 states
    NFA generating a 16 states DFA through the algorithm)
alphabet :: [Label]
alphabet = [Label 0, Label 1]
initStates :: Set.Set State
initStates = Set.fromList [0]
finalStates :: Set.Set State
finalStates = Set.fromList [4]
successors :: AdjacencyList
successors = Map.fromList [(0, [(Label 0, 0), (Label 1, 0), (Label 1, 1)]), (1, [(
    Label 0, 2), (Label 1, 2)]), (2, [(Label 0, 3), (Label 1, 3)]), (3, [(Label 0, 4),
    (Label 1, 4)]), (4, [])]
exampleAutomaton :: Automaton
exampleAutomaton = Automaton successors alphabet initStates finalStates
initStates2 :: Set.Set State
initStates2 = Set.fromList [0]
finalStates2 :: Set.Set State
```

```haskell
finalStates2 = Set.fromList [2]
successors2 :: AdjacencyList
successors2 = Map.fromList [(0, [(Label 0, 1)]), (1, [(Label 1, 2)]), (2, [(Epsilon, 0)])]
exampleAutomaton2 :: Automaton
exampleAutomaton2 = Automaton successors2 alphabet initStates2 finalStates2

intAlphabet :: [Label]
intAlphabet = List.map Label [0..50]

dumbAutomaton :: Int -> Automaton
dumbAutomaton nStates = Automaton adj intAlphabet (Set.singleton 0) (Set.singleton $ nStates - 1)
            where adj = Map.insert 0 [(Label 0, 0), (Label 1, 0), (Label 1, 1)] allButFirstSucc
                  allButFirstSucc = Map.insert (nStates-1) [] allButFirstAndLast
                  allButFirstAndLast = Map.fromList $ List.map (\n -> (n, [(l, r) | l <- intAlphabet, r <- [n+1,n,  n-1]])) [1..nStates-2]

ioDumbAutomaton :: Int -> IO Automaton
ioDumbAutomaton n = return $ dumbAutomaton n
```

/src/Checks.hs

```haskell
module Checks (checkAccept, checkAlphabet) where
import            Automaton (Automaton (..), Label (..), State)
import            Data.Char (ord)
import            Data.List (find)
import qualified Data.Map  as Map

-- checkAlphabet: Check if word agrees with alphabet for a given Automaton
checkAlphabet :: [Char] -> Automaton -> Bool
checkAlphabet (x:xs) dfa@(Automaton _ alph _ _) = n && checkAlphabet xs dfa
        where n = Label (ord x) `elem` alph
checkAlphabet [] _ = True

-- checkAccept: check if word is in a language. Automaton MUST be a DFA
checkAccept :: [Char] -> Automaton -> State -> Bool
checkAccept (x:xs) dfa c = case findTransition x dfa c of
  Just e  -> checkAccept xs dfa n where (_,n) = e
  Nothing -> False
checkAccept [] (Automaton _ _ _ end) c = c `elem` end

findTransition :: Char -> Automaton -> Int -> Maybe (Label, State)
findTransition x (Automaton lst _ _ _) c = case Map.lookup c lst of
 Just es -> find (\y -> fst y == Label (ord x)) es
 Nothing -> Nothing
```

/src/Dictionary.hs

```haskell
module Dictionary (dictNfa) where

import            Automaton (AdjacencyList, Automaton (..), Label (..), State)
import            Data.Char (ord)
import qualified Data.Map  as Map
import qualified Data.Set  as Set
import            WordList  (buildList)

-- addWord: (non thompsons) helper method for dictNFA
addWord :: [Char] -> AdjacencyList -> Set.Set Label -> State -> Set.Set State -> State -> (AdjacencyList, Set.Set Label, Set.Set State, State)
addWord (x:xs) adjList alph fromState finalStates toState = addWord xs newAdjList newAlph toState finalStates newToState where
  t = Label (ord x)
  newToState = toState + 1
  newAlph = Set.insert t alph
  newAdjList = Map.insertWith (++) fromState [(t, toState)] adjList
addWord [] adjList alph lastState finalStates toState = (adjList, alph, newFinals, toState) where
  newFinals = Set.insert lastState finalStates

-- dictNFA: (non thompsons) helper method for buildDictNFA
```

```haskell
dictNFA :: [[Char]] -> AdjacencyList -> Set.Set Label -> State -> Set.Set State ->
    State -> (AdjacencyList, Set.Set Label, Set.Set State)
dictNFA (x:xs) lst alph st fi l = dictNFA xs nlst nalph st nfi nl where
  (nlst, nalph, nfi, nl) = addWord x lst alph 0 fi il
  il = l+1
dictNFA [] lst alph _ fi _ = (lst, alph, fi)

-- makeDictNFA: Build NFA from dict of accepted strings in a language
makeDictNFA :: [[Char]] -> Automaton
makeDictNFA l@(_:_) = Automaton lst alph start fi where
  alph = Epsilon : Set.toList salph
  (lst, salph, fi) = dictNFA l (Map.insert 0 [] Map.empty) Set.empty 0 Set.empty 0
  start = Set.singleton 0
makeDictNFA [] = error "empty dictionary"

dictNfa :: FilePath -> IO Automaton
dictNfa fp = do
    wordList <- buildList fp
    return $ makeDictNFA wordList
```

/src/RandomNfa.hs

```haskell
module RandomNfa (ioRandomNfa) where
import            Automaton     (Label (..))
import qualified Automaton      as A
import qualified Data.List      as List
import qualified Data.Map       as Map
import qualified Data.Set       as Set
import qualified System.Random as Random



-- Inspired from https://hackage.haskell.org/package/random-1.2.1.1/docs/System-Random
    .html
rolls :: Int -> Int -> Int -> [Int]
rolls n maxInt seed = take n . List.unfoldr (Just . Random.uniformR (0, maxInt)) $
    Random.mkStdGen seed

-- Generate a random NFA with a given number of states, an alphabet size, a number of
    final states and a probability
-- (probability that there is an edge with a given label betwen 2 given states)
randomNFA :: Int -> Int -> Int -> Int -> A.Automaton
randomNFA numStates alphabetSize nbFinals proba =
    A.Automaton transitions alphabet initStates finalStates
    where

        intForGen = numStates + alphabetSize + nbFinals + proba

        -- list of states
        states = [0..(numStates-1)]
        -- final states
        finalStates = Set.fromList [(numStates - nbFinals)..(numStates-1)]
        -- alphabet
        alphabet = [Label i | i <- [0..(alphabetSize - 1)]]

        allTransitions = [
            (state1, symbol, state2) |
            state1 <- states,
            state2 <- states,
            symbol <- alphabet ]

        keepTransitions = [ rdInt <= proba | rdInt <- rolls (length allTransitions)
    100 intForGen]

        transitionsList = [ v | (v,keep) <- List.zip allTransitions keepTransitions,
    keep]

        -- turn the list into a map
        transitions :: A.AdjacencyList
        transitions =
            List.foldl'
                (\m (fromS, label, toS) -> Map.insertWith (++) fromS [(label, toS)] m
```

```
    )
                Map.empty
                transitionsList

        -- Generate a random start state
        initStates = Set.singleton 0

ioRandomNfa :: Int -> Int -> Int -> Int -> IO A.Automaton
ioRandomNfa nbStates alphabetSize nbFinals probability =
    return $ randomNFA nbStates alphabetSize nbFinals probability
```

/src/SubsetConstruction.hs

```
module SubsetConstruction (nfaToDfa) where
import qualified Automaton                  as A
import           Control.Parallel.Strategies (parMap, rdeepseq)
import qualified Data.List                  as List
import qualified Data.Map                   as Map
import           Data.Maybe                 (fromMaybe)
import qualified Data.Set                   as Set

exploreLabelFromNFAState :: A.AdjacencyList -> A.Label -> A.State -> [A.State]
exploreLabelFromNFAState nfaAdjacency label state = [s | (l, s) <- edges, l == label]
    where edges = fromMaybe [] (Map.lookup state nfaAdjacency)

exploreLabelFromDFAState :: A.AdjacencyList -> A.Label -> Set.Set A.State -> Set.Set A.State
exploreLabelFromDFAState nfaAdjacency label =
    Set.fromList
    . List.concatMap (exploreLabelFromNFAState nfaAdjacency label)
    . Set.toList

epsilonClosure :: A.AdjacencyList -> Set.Set A.State -> Set.Set A.State
epsilonClosure nfaAdjacency nfaStates   | Set.size nfaStates ==  Set.size explored = nfaStates
                                        | otherwise = epsilonClosure nfaAdjacency explored
                    where explored = Set.union nfaStates (exploreLabelFromDFAState nfaAdjacency A.Epsilon nfaStates)


nextStates :: A.AdjacencyList -> [A.Label] -> [Set.Set A.State] -> [(A.Label, Set.Set A.State, Set.Set A.State)]
nextStates nfaAdjacency alphabet dfaStates =
        parMap rdeepseq
        (\(l, s) -> (l, s, (epsilonClosure nfaAdjacency . exploreLabelFromDFAState nfaAdjacency l) s))
        [(l,s) |
            l <- alphabet,
            s <- dfaStates
        ]

addDfaEdge ::  Set.Set A.State -> (A.DfaStatesMap, A.AdjacencyList, Set.Set A.State, [Set.Set A.State])
                -> (A.Label, Set.Set A.State, Set.Set A.State)
                -> (A.DfaStatesMap, A.AdjacencyList, Set.Set A.State, [Set.Set A.State])
addDfaEdge nfaFinals ((dfaSM, maxIdx), dfaA, dfaF, tV) (l, originS, destS) = case Map.lookup destS dfaSM of
                Nothing | Set.null destS -> ((dfaSM, maxIdx), dfaA, dfaF, tV)
                Nothing -> ((Map.insert destS newState dfaSM, newState), Map.insertWith (++) originState [(l, newState)] dfaA, newDfaF, destS : tV)
                    where newState = maxIdx + 1
                          isFinal = List.any (`Set.member` nfaFinals) (Set.toList destS)
                          newDfaF = if isFinal then Set.insert newState dfaF else dfaF
                Just s -> ((dfaSM, maxIdx), Map.insertWith (++) originState [(l,s)] dfaA, dfaF, tV)
            where originState = case Map.lookup originS dfaSM of
                    Just st -> st
                    Nothing -> error "Couldn't find origin state"

```

```haskell
explore :: A.AdjacencyList -> Set.Set A.State -> [A.Label] -> A.DfaStatesMap -> A.AdjacencyList -> Set.Set A.State -> [Set.Set A.State] -> (A.AdjacencyList, Set.Set A.State)
explore _ _ _ _ dfaAdjacency dfaFinals []                                    = (dfaAdjacency, dfaFinals)
explore nfaAdjacency nfaFinals alphabet dfaStatesMap dfaAdjacency dfaFinals toVisit =
    explore nfaAdjacency nfaFinals alphabet newDfaSM newDfaA newDfaFinals newToVisit
        where   (newDfaSM, newDfaA, newDfaFinals, newToVisit) =
                    List.foldl'
                        (addDfaEdge nfaFinals)
                        (dfaStatesMap, dfaAdjacency, dfaFinals, [])
                        nStates
                nStates = nextStates nfaAdjacency alphabet toVisit

nfaToDfa :: A.Automaton -> A.Automaton
nfaToDfa (A.Automaton nfaAdjacency alphabet inits nfaFinals) = A.Automaton newAdjacency alphabet dfaInits dfaFinals
        where (newAdjacency, dfaFinals) = explore nfaAdjacency nfaFinals alphabet (initDfaStatesMap, 0) initDfaAdjacency Set.empty [inits]
              initDfaStatesMap =  Map.fromList [(inits, 0)]
              initDfaAdjacency =  Map.fromList [(0, [])]
              dfaInits = Set.fromList [0]
```

/src/Thompson.hs

```haskell
module Thompson (regexParser, makeThompsonNFA) where
import           Automaton                             (AdjacencyList,
                                                        Automaton (..), Label (..),
                                                        State)
import           Control.Monad                         (msum)
import           Data.Char                             (ord)
import qualified Data.Map                              as Map
import qualified Data.Set                              as Set
import qualified Text.ParserCombinators.Parsec         as P
import qualified Text.ParserCombinators.Parsec.Expr    as PE

data Node = Concat !Node !Node | Star !Node | Or !Node !Node | Character !Int deriving (Show)

regexParser :: P.Parser Node
regexParser = PE.buildExpressionParser opTable base
  where
    opTable = [   [ PE.Postfix (P.char '*' >> return Star)]
                , [ PE.Infix (return Concat) PE.AssocLeft]
                , [ PE.Infix (P.char '|' >> return Or) PE.AssocLeft]
              ]
    base = msum [Character . ord <$> P.noneOf "()*|", parens regexParser]
    parens = P.between (P.char '(') (P.char ')')

-- Build NFA from regex AST following Thompson's Algorithm
makeThompsonNFA :: Either a Node -> Automaton
makeThompsonNFA ((Right ast)) = Automaton table alphabet start end where
  (table, _) = thompsons ast 1 0 1
  alphabet = Set.toList (buildAlph ast Set.empty)
  start = Set.singleton 1
  end = Set.singleton 0
makeThompsonNFA (Left _) = error "Bad AST"

-- buildAlph: Build alphabet from regex AST. Helper method for makeThompsonNFA
buildAlph :: Node -> Set.Set Label -> Set.Set Label
buildAlph (Concat r l) alph  = Set.union (buildAlph r alph) (buildAlph l alph)
buildAlph (Star l) alph      = buildAlph l alph
buildAlph (Or r l) alph      = Set.union (buildAlph r alph) (buildAlph l alph)
buildAlph (Character x) alph = Set.insert (Label x) alph

-- thompsons: Build adjacency list from regex AST. Helper method for makeThompsonNFA
thompsons :: Node -> State -> State -> State -> (AdjacencyList, State)
thompsons (Character x) q f l = (Map.fromList [(q, [(Label x, f)])], l)
thompsons (Concat s t) q f l = (Map.union smap tmap, lt) where
  (smap, ls) = thompsons s q i ln
  (tmap, lt) = thompsons t i f ls
  i = l + 1
```

```
  ln = l + 1
thompsons (Or s t) q f l = (Map.union (Map.union smap tmap) omap, lt) where
  omap = Map.fromList [(q, [(Epsilon, si), (Epsilon, ti)]), (sf, [(Epsilon, f)]), (tf, [(Epsilon, f)])]
  (smap, ls) = thompsons s si sf ln
  (tmap, lt) = thompsons t ti tf ls
  si = l+1; sf = l+2
  ti = l+3; tf = l+4
  ln = l+4
thompsons (Star s) q f l = (Map.union smap stmap, ls) where
  stmap = Map.fromList [(q, [(Epsilon, si), (Epsilon, f)]), (sf, [(Epsilon, si), (Epsilon, f)])]
  (smap, ls) = thompsons s si sf ln
  si = l+1; sf = l+2
  ln = l+2
```

/src/WordList.hs

```
module WordList (buildList) where
import           Data.Char    (isAlpha)
import qualified Data.Set     as Set
import qualified Data.Text    as T
import           Data.Text.IO as TIO (readFile)

buildList :: FilePath -> IO [[Char]]
buildList filename = do
 h <- TIO.readFile filename
 let l = T.words h
 let lnorm = map normalize l
 let lnormset = Set.fromList lnorm
 let lnormunique = Set.toList lnormset
 return lnormunique

normalize :: T.Text -> [Char]
normalize string = [ x | x <- a, isAlpha x ] where a = T.unpack (T.toLower string)
```

/Setup.hs

```
import Distribution.Simple
main = defaultMain
```

/stack.yaml

```
# This file was automatically generated by 'stack init'
#
# Some commonly used options have been documented as comments in this file.
# For advanced use and comprehensive documentation of the format, please see:
# https://docs.haskellstack.org/en/stable/yaml_configuration/

# Resolver to choose a 'specific' stackage snapshot or a compiler version.
# A snapshot resolver dictates the compiler version and the set of packages
# to be used for project dependencies. For example:
#
# resolver: lts-3.5
# resolver: nightly-2015-09-21
# resolver: ghc-7.10.2
#
# The location of a snapshot can be provided as a file or url. Stack assumes
# a snapshot provided as a file might change, whereas a url resource does not.
#
# resolver: ./custom-snapshot.yaml
# resolver: https://example.com/snapshots/2018-01-01.yaml
system-ghc: true
resolver: ghc-8.10.7

# User packages to be built.
# Various formats can be used as shown in the example below.
#
# packages:
# - some-directory
# - https://example.com/foo/bar/baz-0.0.2.tar.gz
#   subdirs:
#   - auto-update
```

```
#   - wai
packages:
  - .
# Dependency packages to be pulled from upstream that are not in the resolver.
# These entries can reference officially published versions as well as
# forks / in-progress versions pinned to a git hash. For example:
#
extra-deps:
  - random-1.2.1.1
  - splitmix-0.1.0.4
  - parallel-3.2.2.0
# - acme-missiles-0.3
# - git: https://github.com/commercialhaskell/stack.git
#   commit: e7b331f14bcffb8367cd58fbfc8b40ec7642100a
#
# extra-deps: []

# Override default flag values for local packages and extra-deps
# flags: {}

# Extra package databases containing global packages
# extra-package-dbs: []

# Control whether we use the GHC we find on the path
# system-ghc: true
#
# Require a specific version of stack, using version ranges
# require-stack-version: -any # Default
# require-stack-version: ">=2.9"
#
# Override the architecture used by stack, especially useful on Windows
# arch: i386
# arch: x86_64
#
# Extra directories used by stack for building
# extra-include-dirs: [/path/to/dir]
# extra-lib-dirs: [/path/to/dir]
#
# Allow a newer minor version of GHC than the snapshot specifies
# compiler-check: newer-minor
```

/stack.yaml.lock

```
# This file was autogenerated by Stack.
# You should not edit this file by hand.
# For more information, please see the documentation at:
#   https://docs.haskellstack.org/en/stable/lock_files

packages:
- completed:
    hackage: random-1.2.1.1@sha256:dea1f11e5569332dc6c8efaad1cb301016a5587b6754943a49f9de08ae0e56d9,6541
    pantry-tree:
      sha256: 646ee77fe01178837ee928b61ae8653dcf190e9b5353ebebd094079c77a18b76
      size: 1528
  original:
    hackage: random-1.2.1.1
- completed:
    hackage: splitmix-0.1.0.4@sha256:804e2574bc7e32d08cbab91e47ee6287b4df7d50851d73f9e778f94a9a7814c7,6521
    pantry-tree:
      sha256: b56f706c092dc0ac4875ef45bd18719386358c56667f6b604a733b66f9e4657f
      size: 1519
  original:
    hackage: splitmix-0.1.0.4
- completed:
    hackage: parallel-3.2.2.0@sha256:6edd5a06938cea3d28b406d5231683f89737e854af144a8800aa69e1eee785e0,1821
    pantry-tree:
      sha256: 6ec36425356925d6d9042769a29ab4ec2aa69c2a7161c49ff18a9a77c1d957b1
      size: 392
  original:
```

```
    hackage: parallel-3.2.2.0
snapshots: []
```

/package.yaml

```
name: subset-construction
version: 0.1.0.0
github: "AlexisGado/subset-construction"
license: BSD3
author: "Alexis Gadonneix"
maintainer: "ag4625@columbia.edu"
copyright: "2022 Alexis Gadonneix"

extra-source-files:
  - README.md
  - CHANGELOG.md
  - assets/**

# Metadata used when publishing your package
# synopsis:            Short description of your package
# category:            Web

# To avoid duplicated efforts in documentation and dealing with the
# complications of embedding Haddock markup inside cabal files, it is
# common to point users to the README.md file.
description: Please see the README on GitHub at <https://github.com/githubuser/subset-construction#readme>

dependencies:
  - base >= 4.7 && < 5
  - containers
  - random
  - parsec
  - parallel
  - deepseq
  - text
  - mtl

ghc-options:
  - -Wall
  - -Wcompat
  - -Widentities
  - -Wincomplete-record-updates
  - -Wincomplete-uni-patterns
  - -Wmissing-export-lists
  - -Wmissing-home-modules
  - -Wpartial-fields
  - -Wredundant-constraints

library:
  source-dirs: src

executables:
  subset-construction-exe:
    main: Main.hs
    source-dirs: app
    ghc-options:
      - -threaded
      - -rtsopts
      - -eventlog
      - -with-rtsopts=-N
      - -O2
    dependencies:
      - subset-construction

tests:
  subset-construction-test:
    main: Spec.hs
    source-dirs: test
    ghc-options:
      - -threaded
      - -rtsopts
```

```
      - -with-rtsopts=-N
    dependencies:
      - subset-construction
```

/app/Main.hs

```haskell
module Main (main) where
import            Automaton          (Automaton (..))
import qualified Data.Map            as Map
import            RandomNfa          (ioRandomNfa)
import qualified SubsetConstruction as SC

usedAutomaton :: IO Automaton
-- random NFA with control over density and size
usedAutomaton = ioRandomNfa 600 20 10 50

-- NFA from a list of words
-- usedAutomaton = dictNfa "assets/words.txt"

-- NFA from a regular expression
-- usedAutomaton = return $ makeThompsonNFA $ P.parse regexParser "" "a*(a|b)b*"

-- Some simple examples
-- usedAutomaton = return exampleAutomaton
-- usedAutomaton = return exampleAutomaton2

-- a simple linear automaton to test scale
-- usedAutomaton = ioDumbAutomaton 300


main :: IO ()
main = do
    nfa <- usedAutomaton
    let Automaton nfaAdj _ _ _ = nfa
    let Automaton dfaAdj _ _ _ = SC.nfaToDfa nfa
    print $ Map.size nfaAdj
    print $ Map.size dfaAdj
```