Supplementary materials for algorithms in FlexAmata

Algorithm 1: n-bit to 1-bit converter

```
Input  : an automaton A with n-bits per symbol
Output : an automaton B with 1-bit per symbol
Initialize mapping M      // for mapping states from A to B
forall states s in A do
    Initialize new state t
    add t to B
    add mapping s to t in B
forall edges e in A with source src and destination dst do
    forall symbols sym on e do
        Convert sym to n bits binary b_{n-1}b_{n-2},...,b_0
        Create a new path labeled with b_{n-1}b_{n-2},...,b_0 from M[src]
          to M[dst]
```

Algorithm 1: This algorithm converts an n-bit automaton A to its equivalent 1-bit-automaton. Our algorithm first copies all the states in A into B and keeps the mapping in M. For each symbol on every edge of A, we generate the equivalent n-bit binary representation of symbols (padding with zeros if necessary). This n-bits will be used later to label a new path (with length n) in B to connect the copies of current edge's sources and destinations in B.

Algorithm 2: the input automaton A will be converted to an equivalent automaton B, and each symbol has bps bits. Our algorithm basically solves a graph processing problem that looks for all paths of length bps in A, starting from a subset of states as sources. While using a simple backtracking algorithm to find all the paths starting from any arbitrary state can solve the problem, it suffers from recalculating paths that already have been traversed previously. To solve this issue, we use a dynamic programming approach combined with backtracking algorithm to solve this problem more efficiently and reuse previously calculated paths. Our algorithm keeps a global dictionary D to save all the mid results and partial paths (paths shorter than bps) as soon as they have been calculated. Our $strider$ function in Algorithm 1 receives a source state with any arbitrary bps and greedily looks in to D to check if we have ever passed the source previously, even for reaching shorter paths than bps. If yes, it greedily picks the biggest jump, which we had previously calculated as a partial paths and jumps to their destinations, and it continues calculating of remaining part of the path from there. If D is empty from any information regrading the source, it starts from the direct approach and use the backtracking approach. It recursively call the $strider$ function again, but with a reduced path length from the direct neighbors, and results calculated using the backtracking will be placed in D to be reused again if necessary.

Algorithm 2: 1-bit to m-bit converter

```
Input  : bitwise automaton A
Input  : target number of bits per symbol bps
Output : an automaton B with bps bits per symbol
Initialize global dictionary D         // mid results container
Initialize stack Q                     // for DFS visiting policy
Initialize map M             // for mapping states from A to B
Push start states in Q
Copy start states into B and add mapping into M
while Q is not empty do
    curr_state = Q.pop()
    Call strider(curr_state, bps)
    forall descendant d in D[(curr_state, bps)] with symbol list s do
        if d was not met before then
            Q.push(d)
            Add copy of d to B and add mapping in M
        B.add_edge(M[curr_state], M[d]) with label s
Function strider(src, t_bps) is
    if (src, t_bps) in D then
        /* It has previously been calculated.         */
        return
    else if t_bps == 1 then
        /* looking for all the direct children        */
        Initialize a new dictionary temp_d with default value [ ]
        /* key: child state, value: list of symbols
           on edges from src to the child             */
        forall child t of src with symbol s on edge do
            temp_d[t].add(s)
        D[(src, t_bps)] = temp_d
        return
    else if there exists an m which m < t_bps and (src, m) in D then
        /* we have previously calculated all paths
           from src with length m                     */
        Initialize a new dictionary temp_d with default value [ ]
        Find biggest m that (src, m) is in D
        Iterate through all keys j in D[(src, m)] and call
          strider(j, t_bps - m) for each of them
        Combine results of D[(src, m)] and D[(j, t_bps - m)] and
          place in temp_d
        D[(src, t_bps)] = temp_d
        return
    else
        /* This state has not been reached yet        */
        Initialize a new dictionary temp_d with default value [ ]
        forall neighbors t1 of src with symbol s1 do
            Call strider(t1, t_bps - 1))
            forall reached nodes r in D[(t, t_bps - 1)] with symbol
              s2
             do
                temp_d[r].add(s1 * 2^{t_bps} + s2)
        D[(src, t_bps)] = temp_d
        return
```