Verification of Reactive Systems

Lecture notes for Lecture 2: Revision of Graphs and Automata Theory

Johannes Kloos

May 2, 2014

1 Graphs and Graph Algorithms

There are two types of graphs: *directed graphs* and *undirected graphs*. A directed graph (short *digraph*)is given by a tuple

$$G = (V, E) \text{ where } E \subseteq V \times V.$$

The set V is known as the set of *vertices*, and the set E as the set of *edges*. Directed graphs can be represented visually as in Figure 1.

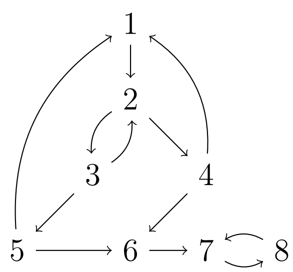
Let a digraph $G = (V, E)$ be given. For two vertices $u, v \in V$, write $u \to v$ iff $(u, v) \in E$. Given a vertex v, define the sets

$$\operatorname{pred}(v) := \{w \mid w \to v\} \text{ and } \operatorname{succ}(v) := \{w \mid v \to w\},$$

known as the set of *predecessors* and the set of *successors*. The *in-degree* of a vertex v is defined as $\operatorname{indeg} v := |\operatorname{pred}(v)|$, and the *out-degree* of v as $\operatorname{outdeg} v := |\operatorname{succ}(v)|$. In the example, the following is true:

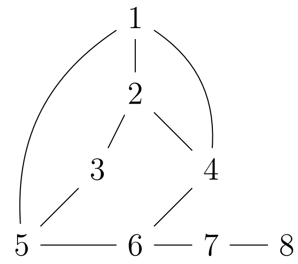
$$\operatorname{pred}(2) = \{1\} \qquad \operatorname{succ}(2) = \{3, 4\} \qquad \operatorname{indeg}(2) = 1 \qquad \operatorname{outdeg}(2) = 2$$

$$\operatorname{pred}(6) = \{4, 5\} \qquad \operatorname{succ}(6) = \{7\} \qquad \operatorname{indeg}(6) = 2 \qquad \operatorname{outdeg}(6) = 1$$



$$\begin{aligned} V &= \{1, 2, 3, 4, 5, 6, 7, 8\} \\ E &= \{(1, 2), (2, 3), (2, 4), (3, 2), (3, 5), (4, 1), \\ &\qquad (4, 6), (5, 1), (5, 6), (6, 7), (7, 8), (8, 7)\} \end{aligned}$$

Figure 1: An example of a directed graph.



$$V = \{1, 2, 3, 4, 5, 6, 7, 8\}$$
$$E = \{\{1, 2\}, \{2, 3\}, \{2, 4\}, \{3, 5\}, \{4, 1\}, \{4, 6\}, \{5, 1\}, \{5, 6\}, \{6, 7\}, \{7, 8\}, \}$$

Figure 2: An example of an undirected graph.

An undirected graph is given by a tuple

$$G = (V, E) \text{ where } E \subseteq \{\{u, v\} \mid u, v \in V \text{ and } u \neq v\}.$$

They can also be represented visually, as in Figure 2. Let an undirected graph $G = (V, E)$ be given. Similar to the case of digraphs, define the set of *neighbors*

$$\mathrm{neigh}(v) := \{w \mid \{v, w\} \in E\}$$

and the *degree* of a vertex v as $\deg(v) = |\,\mathrm{neigh}(v)|$. One may write $u - v$ for any $u, v \in V$ such that $\{u, v\} \in E$.

For both directed and undirected graphs, the important notion of *path* can be defined.

Definition 1. Let $G = (V, E)$ be a directed graph.

1. A sequence $v_1, \ldots, v_n$ of vertices is called a *path* if, for all $i = 1, \ldots, n-1$, $v_i \to v_{i+1}$.
2. For two vertices $v, w \in V$, there is a path of length n, written $v \to^n w$, iff there is a path $v_1, \ldots, v_n$ such that $v = v_1$ and $w = v_n$.
3. For two vertices $v, w \in V$, there is a path from v to w, written $v \to^* w$, if there is an $n \geq 0$ such that $v \to^n w$.

If $v \to^* w$, one may also say that w *is reachable from* v.

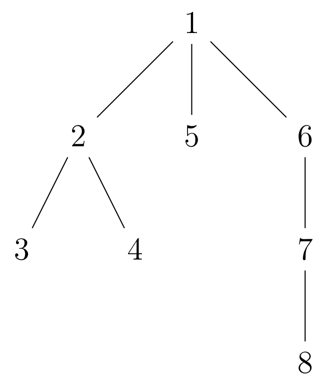
The corresponding definitions for undirected graphs are similar, except that in the definition of a path, we demand that $v_i - v_{i+1}$ for $i = 1, \ldots, n-1$.

A *tree* is recursively defined as a set V of vertices such that

1. There is a special node $v_0 \in V$, called the *root* of V.
2. There are sets $V_1, \ldots, V_m$, with $m \geq 0$, such that $V \setminus \{v_0\} = V_1 \cup \cdots \cup V_m$, the V_i are disjoint and each V_i is again a tree.

The sets V_i are called the *subtrees* of the root (compare [1], Section 2.3).

A tree can be represented in a natural way as a directed graph: Define $E = \{(v, v') \in V \times V \mid v' \text{ is the root of a subtree of } v\}$. An example of a tree is given in Figure 3. It



Tree structure:
$\{3\}$, $\{4\}$, $\{5\}$ and $\{8\}$ form trees with their unique element as root and no subtrees.
$\{7, 8\}$ forms a tree with root 7 and single subtree $\{8\}$, while $\{2, 3, 4\}$ forms a tree with root 2 and subtrees $\{3\}$ and $\{4\}$. $\{6, 7, 8\}$ forms a tree with root 6 and single subtree $\{7, 8\}$.
Finally, $\{1, 2, 3, 4, 5, 6, 7, 8\}$ is a tree with root 1 and subtrees $\{2, 3, 4\}$, $\{5\}$ and $\{6, 7, 8\}$.

Figure 3: An example of a tree.

can be shown ([1], exercise 2.3.3) that in this graph, there is a unique path from the root v of V to every vertex $v' \in V$.

Obviously, each directed graph (V, E) gives rise to an undirected graph (V, E') with $E' = \{\{v, w\} \mid v \to w\}$ by "forgetting the direction" of the edges (compare the graphs in Figure 1 and 2). Conversely, an undirected graph (V, E) induces a directed graph $(V, \{(v, w), (w, v) \mid v - w\})$.

1.1 Graph traversal

One common algorithmic question when dealing with graphs is the following: Given a (directed or undirected) graph $G = (V, E)$, a vertex $v \in V$ and some set $P \subseteq V$, is there are vertex w such that $v \to^* w$ (respectively $v -^* w$) and $w \in P$? In the following, the algorithms will be presented in terms of directed graphs; the corresponding algorithms for undirected graphs are similar.

There are two main algorithms that differ in their search strategy. *Depth-first search*, short *DFS*, works by constantly extending a potential path with new vertices and backtracking if no extension is possible. *Breadth-first search*, short *BFS* works by considering vertices in "layers" around the initial vertex: first those reachable in one step, then those reachable in two steps and so on.

The DFS algorithm can be given as follows:

// The graph $G = (V, E)$ is given, as is the set $P \subseteq V$.
function DFS(v)
    // The function returns either a vertex v' with $v' \in P$,
    // or `Fail` if no such vertex exists
    mark v
    if $v \in P$ **then**
        return v
    end if
    for $v' \in \mathrm{succ}(v)$ where v' is not marked **do**
        if `DFS`(v') returns a state v'' **then**

Figure 4: An example execution of DFS

Action	Marked nodes	Nodes to visit	Call stack
Call DFS(3)	$\varnothing$	–	
Mark 3	$\{3\}$	–	
Iterate over succ(3) = $\{2, 5\}$	$\{3\}$	$\{2, 5\}$	
Call DFS(5)	$\{3\}$	–	$3/\{2\}$
Mark 5	$\{3, 5\}$	–	$3/\{2\}$
Iterate over succ(5) = $\{1, 6\}$	$\{3, 5\}$	$\{1, 6\}$	$3/\{2\}$
Call DFS(6)	$\{3, 5\}$	–	$3/\{2\}; 5/\{1\}$
Mark 6	$\{3, 5, 6\}$	–	$3/\{2\}; 5/\{1\}$
Iterate over succ(6) = $\{7\}$	$\{3, 5, 6\}$	$\{7\}$	$3/\{2\}; 5/\{1\}$
Call DFS(7)	$\{3, 5, 6\}$	–	$3/\{2\}; 5/\{1\}; 6/\varnothing$
Mark 7	$\{3, 5, 6, 7\}$	–	$3/\{2\}; 5/\{1\}; 6/\varnothing$
Iterate over succ(7) = $\{8\}$	$\{3, 5, 6, 7\}$	$\{8\}$	$3/\{2\}; 5/\{1\}; 6/\varnothing$
Call DFS(8)	$\{3, 5, 6, 7\}$	–	$3/\{2\}; 5/\{1\}; 6/\varnothing; 7/\varnothing$
Mark 8	$\{3, 5, 6, 7, 8\}$	–	$3/\{2\}; 5/\{1\}; 6/\varnothing; 7/\varnothing$
Iterate over succ(8) = $\{7\}$	$\{3, 5, 6, 7, 8\}$	$\{7\}$	$3/\{2\}; 5/\{1\}; 6/\varnothing; 7/\varnothing$
Skip 7 – it is marked	$\{3, 5, 6, 7, 8\}$	$\varnothing$	$3/\{2\}; 5/\{1\}; 6/\varnothing; 7/\varnothing$
Return "fail" to DFS(7)	$\{3, 5, 6, 7, 8\}$	$\varnothing$	$3/\{2\}; 5/\{1\}; 6/\varnothing$
Return "fail" to DFS(6)	$\{3, 5, 6, 7, 8\}$	$\varnothing$	$3/\{2\}; 5/\{1\}$
Return "fail" to DFS(5)	$\{3, 5, 6, 7, 8\}$	$\{1\}$	$3/\{2\}$
Call DFS(1)	$\{3, 5, 6, 7, 8\}$	–	$3/\{2\}$
Return 1 to DFS(3) ($1 \in P$)	$\{3, 5, 6, 7, 8\}$	–	$3/\{2\}$
Return 1	$\{3, 5, 6, 7, 8\}$	–	

            return v''
        end if
    end for
    return `Fail`
end function

As an example, DFS can be used to check whether the node 1 can be reached from the node 3 in the digraph from Figure 1. The execution trace is presented in Figure 4 in a compact format: Each line of the table contains an action that is performed, and the state of the execution after the action finishes. It contains the set of nodes that have been marked so far (second column), the nodes that still need to be visited in the inner loop of the DFS function (third column), and the call stack. The elements of the call stack are of the form v/s, where v denotes the node given in the corresponding call to DFS, and s the set of nodes that still needs to be visited in this call.

A side effect of the DFS algorithm is the following: If the algorithm is run on a vertex v and returns `Fail`, all states reachable from v are marked. In particular, to compute the states reachable from v, call DFS(v) with $P = \varnothing$. Then the set of marked states is exactly the set of states reachable from v.

The DFS algorithm can be extended so that if DFS(v) returns v', it also returns an actual path from v to v'. There is no guarantee that a shortest path will be found: A DFS from 1 trying to reach 5 may well return the path $1 \to 2 \to 3 \to 5$ instead of the shorter path $1 \to 4 \to 5$.

The BFS algorithm works iteratively, using a queue Q:

```
// The graph G = (V, E) is given, as is the set P ⊆ V.
function BFS(v)
    // The function returns either a vertex v' with v' ∈ P,
    // or Fail if no such vertex exists
    Q ← new Queue
    Add v to Q
    while Q is not empty do
        v' ← first(Q)
        Remove v' from Q
        if v' is marked then
            continue
        end if
        Mark v'
        if v' ∈ P then
            return v'
        end if
        for all v'' such that v' → v'' do
            Add v'' to Q
        end for
    end while
    return Fail
end function
```

In Figure 5, an example execution of BFS to find a path from vertex 3 to vertex 1 in the graph of Figure 1 is presented. Again, the execution is presented in a tabular format, listing the action and the state of the algorithm after the action has finished.

Again, this algorithm can be extended to return an actual path. It is easy to show that such a path is always a shortest path, i.e., there is no shorter path.

Theorem 1 (Run-time Complexity of BFS and DFS)**.** *Both BFS and DFS run in time $O(|E|)$.*

Proof sketch for BFS: Let $W \subseteq V$ be the set of states that gets marked during the execution. One can prove that each state $w \in W$ is visited at most $\operatorname{indeg}(w)$ times, and that the inner loop is reached exactly $|W|$ or $|W| - 1$ times. For a state w, the inner loop runs $\operatorname{outdeg}(w)$ times. Therefore, the outer loop runs at most $\sum_{w \in W} \operatorname{indeg}(w)$ times, and the inner loop runs at most $\sum_{w \in W} \operatorname{outdeg}(w)$ times. Since $\sum_{w \in W} \operatorname{indeg}(w) + \sum_{w \in W} \operatorname{outdeg}(w) \leq 2|E|$, the claim follows.

Figure 5: An example of BFS

Action	v'	Marked nodes	Contents of Q
Initialize	–	$\varnothing$	$[3]$
Take 3 from Q	3	$\varnothing$	$[]$
Mark 3	3	$\{3\}$	$[]$
Add succ(3) $= \{2, 5\}$ to Q	3	$\{3\}$	$[2, 5]$
Take 2 from Q	2	$\{3\}$	$[5]$
Mark 2	2	$\{2, 3\}$	$[5]$
Add succ(2) $= \{3, 4\}$ to Q	2	$\{2, 3\}$	$[5, 3, 4]$
Take 5 from Q	5	$\{2, 3\}$	$[3, 4]$
Mark 5	$\{2, 3, 5\}$	$[3, 4]$	
Add succ(5) $= \{1, 6\}$ to Q	5	$\{2, 3, 5\}$	$[3, 4, 1, 6]$
Take 3 from Q	3	$\{2, 3, 5\}$	$[4, 1, 6]$
Continue (3 is marked)	3	$\{2, 3, 5\}$	$[4, 1, 6]$
Take 4 from Q	4	$\{2, 3, 5\}$	$[1, 6]$
Mark 4	$\{2, 3, 4, 5\}$	$[1, 6]$	
Add succ(4) $= \{1, 6\}$ to Q	4	$\{2, 3, 4, 5\}$	$[1, 6, 1, 6]$
Take 1 from Q	1	$\{2, 3, 4, 5\}$	$[6, 1, 6]$
Return 1 ($1 \in P$)	1	$\{2, 3, 4, 5\}$	$[6, 1, 6]$

1.2 Distances and shortest paths

For a given graph G, the *distance* between two nodes v and w, written $d(v, w)$, is defined to be the length of the shortest path between v and w. If there is no path at all between v and w, we write $d(v, w) = \infty$. Obviously, $d(v, v) = 0$, and the triangle inequality holds: For $u, v, w \in V$, $d(u, w) \leq d(u, v) + d(v, w)$.

To calculate the distance between two vertices of an arbitrary graph, a variation of BFS can be used:

// A graph $G = (V, E)$ is given
function DISTANCE((v, w))
  $d[v] \leftarrow 0$
  for $u \in V \setminus \{v\}$ **do**
    $d[u] \leftarrow \infty$
  end for
  $Q \leftarrow$ **new** `Queue`
  Add v to Q
  while Q is not empty **do**
    $u \leftarrow$ `first`(Q)
    Remove u from Q
    if $u = w$ **then**
      return $d[u]$

$$
\begin{tikzcd}
1 \arrow[r] \arrow[d] & 2 \arrow[r] & 4 \\
3 \arrow[r] \arrow[d] \arrow[urr] & 5 \arrow[r] & 6 \\
7 \arrow[r] & 8 \arrow[r] & 9 \arrow[u]
\end{tikzcd}
$$

Figure 6: An example of a directed acyclic graph.

```
        end if
        for u' ∈ succ(u) do
            if d[u'] = ∞ then
                d[u'] = 1 + d[u]
                Add u' to Q
            end if
        end for
    end while
    return ∞
end function
```

It is straightforward to modify this algorithm so that it returns the array $d[u]$ that gives the distance from v to all $u \in V$, i.e., $d[u] = d(v, u)$ for all $u \in V$. In either case, the algorithm runs in time $O(|E|)$.

1.3 Cycles, topological order and topological sorting

Another important concept in graphs is the notion of a *cycle*. A cycle is a path $v_1, \ldots, v_n$ such that $v_1 = v_n$; as an example, the path $1 \to 2 \to 3 \to 5 \to 1$ in the graph of Figure 1 is a cycle.

A directed graph without cycles is called a *directed acyclic graph*, short *DAG*. The graph in Figure 1 is not a DAG. On the other hand, every tree is a DAG, and so is the graph in Figure 6.

A related concept is topological order. Let $G = (V, E)$ be a directed graph, and $\ell : V \to \{1, \ldots, |V|\}$ a bijective function that labels the nodes. We say that ℓ induces a topological order if, for all $v, w \in V$ such that $v \to w$, $\ell(v) < \ell(w)$.

Theorem 2. *A directed graph is a DAG if and only if it can be topologically ordered.*

Proof: See any algorithms textbook.

The following algorithm, based on a variant of DFS, will either produce a cycle or a topological order.

```
// The graph G = (V, E) is given.
// Output: Either "cycle: v_1, ..., v_n" or
// "order: ℓ", a function that induces a topological order.
function TopoSort
    for v ∈ V do
```

```
        Color v white.
    end for
    ℓ := new array(|V|)
    i ← 0
    while There is a state v that is colored white do
        r ← visit(v, i)
        if r is "cycle: v_1, ..., v_n" then
            return "cycle: v_1, ..., v_n"
        end if
        // r is now an integer.
        i ← r
    end while
end function
function VISIT((v, i))
    if v is colored black then
        return i
    else if v is colored gray then
        return "cycle: v"
    end if
    Color v gray.
    ℓ[v] ← i
    i ← i + 1
    for v′ ∈ succ(v) do
        r ← visit(v, i)
        if r is "cycle: v_2, ..., v_n" then
            return "cycle: v, v_2, ..., v_n"
        end if
        i ← r
    end for
    Color v black.
end function
```

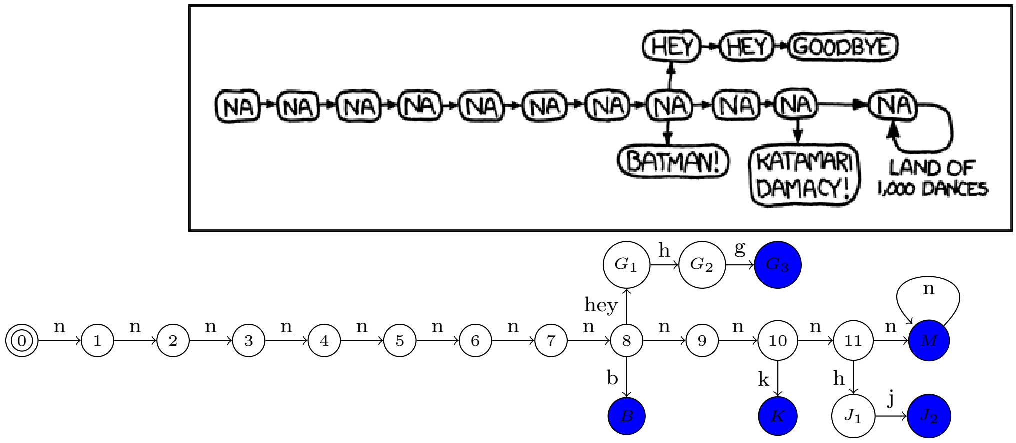
This algorithm has complexity $O(|V| + |E|)$.

1.4 Strongly connected components

Let $G = (V, E)$ be a graph and $C \subseteq V$ be a set of vertices. The *induced graph* of C is $G' = (C, E \cap C \times C)$; it is the graph that can be obtained by removing all vertices and edges from G that are not in C, respectively have an endpoint not in C.

C is said to be *strongly connected* if, for every $v, w \in C$, $v \to^* w$ and $w \to^* v$ in the inducted graph. C is a *strongly connected component* if it is strongly connected, and there is no larger set $C' \supsetneq C$ such that C' is strongly connected.

In the graph in Figure 1, there are three strongly connected components: $\{1, 2, 3, 4, 5\}$, $\{6\}$ and $\{7, 8\}$. There are some further strongly connected subsets, like $\{1, 2, 4\}$ or $\{2, 3\}$.



$$
\begin{aligned}
\Sigma &= \{b, g, h, j, k, n\} \\
Q &= \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, M, B, K, J_1, J_2, G_1, G_2, G_3\} \\
q_0 &= 0 \\
F &= \{M, B, K, J_2, G_3\} \\
\delta &= \{(q_0, n, q_1), \ldots, (q_{10}, n, q_{11}), (q_{11}, n, M), (M, n, M), (q_{11}, h, J_1), (J_1, j, J_2), \\
&\qquad (q_{10}, k, K), (q_8, h, G_1), (G_1, h, G_2), (G_2, g, G_3), (g_8, b, B)\} \\
M &= (Q, \Sigma, \delta, q_0, F)
\end{aligned}
$$

Figure 7: An example of an FSM. It models the language given in the picture (taken from XKCD, see `http://xkcd.com/851/`) on top. The letters of the alphabet Σ are taken as abbreviations of the strings in the picture: n for "na", h for "hey" and so on.

A graph can be decomposed into its strongly connected components in linear time, i.e., time $O(|V| + |E|)$. One algorithm that performs this decomposition is Tarjan's algorithm, based on DFS similar to the topological sorting algorithm above.

2 Automata Theory

A *finite-state machine* or *finite-state automaton*, short *FSM*, is a tuple $M = (Q, \Sigma, \delta, q_0, F)$. Q is a finite set of *states*, and Σ is a finite set of *symbols* called the *alphabet*. The state $q_0 \in Q$ is the *initial state*, and the set $F \subseteq Q$ is the set of *final states*. Finally, $\delta \subseteq Q \times \Sigma \cup \{\epsilon\} \times Q$ (where $\epsilon \notin \Sigma$) is the *transition relation*.

FSMs can be visualized as labeled graphs: (Q, Σ, δ) induces a directed (multi-)graph with labeled edges, and some vertices (namely, q_0 and the elements of F) are marked as initial and/or final. Figure 7 gives an example.

Define by Σ^* the set of *words* over the alphabet Σ: $\Sigma^* = \{w_1 \cdots w_n \mid w_1, \ldots, w_n \in \Sigma\}$.

ϵ is identified with the empty word, i.e., a word $w_1 \dots w_n$ with $n = 0$. The operation $\cdot$ stands for concatenation, i.e., $(u_1 \cdots u_m) \cdot (v_1 \cdots v_n) = u_1 \cdots u_m v_1 \cdots v_n)$.

FSMs are interpreted as devices that *accept* or *generate* words from a certain subset of Σ^*, known as the *language* of the automaton. The following definition makes this precise.

Definition 2. Let a FSM $M := (Q, \Sigma, \delta, q_0, F)$ be given.

1. Let $w \in \Sigma^*$ be a word, $w_1, \dots, w_n \in \Sigma \cup \{\epsilon\}$ where $w = w_1 \cdot w_2 \cdots w_n$ and $q_0, \dots, q_n \in$ be states.

   If $(q_{i-1}, w_i, q_i) \in \delta$ for all $i = 1, \dots, n$, we say that the q_i and the w_i form a *path*.

2. Let q, q' and $w = w_1 \cdots w_n$ be given.

   If there are $q_0, \dots, q_n \in Q$ such that $q_0 = q$, $q_n = q'$ and the q_i and w_i form a path, we say that there is a path from q to q' for w, written $q \xrightarrow{w} q'$.

3. We say that M *accepts* $w \in \Sigma^*$ iff there is a state $q_f \in F$ such that $q_0 \xrightarrow{w} q_f$.

   The *language* of M is the set $L(M) := \{w \in \Sigma^* \mid M \text{ accepts } w\}$.

   Two machines M_1 and M_2 are *language-equivalent* if $L(M_1) = L(M_2)$.

The FSM from Figure 7 has the language

$$L(M) = \{nnnnnnnnb, nnnnnnnnhhg, nnnnnnnnnnk, nnnnnnnnnnnhj\} \cup \{n^i \mid i \geq 12\}$$

For an FSM M, the set of *reachable states* is defined to be the states q such that there is a word $w \in \Sigma^*$ such that $q_0 \xrightarrow{w} q$. Given two FSMs $M_1 = (Q_1, \Sigma, \delta_1, q_0, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, q_0, F_2)$, both having the same set R of reachable states, if $\delta_1(q, \sigma, q') \Leftrightarrow \delta_2(q, \sigma, q')$ for all $q, q' \in R$ and $\sigma \in \Sigma$, and $F_1 \cap R = F_2 \cap R$, then $L(M_1) = L(M_2)$. Two automata with this property are said to *have the same reachable part*.

2.1 Determinism

One important subclass of FSM are *deterministic automata*, short DFA. Formally, a DFA is an FSM $(Q, \Sigma, \delta, q_0, F)$ where $\delta \in Q \times \Sigma \to Q$, i.e., δ is a function that takes a state and a symbol and yields a state. In particular, in a DFA, there are no ϵ-transitions (i.e., members of δ that are labeled with an ϵ-symbol), and for each state q and symbol σ, there is at most one state q' such that $q \overset{\sigma'}{q}$. Note that the FSM in Figure 7 is actually a DFA.

For every FSM M, there is a DFA D such that $L(M) = L(D)$. It can be constructed using the *subset construction*. Here and in the following sections, the construction will be first given as the description of an automaton, and then as an algorithm. While they may give different automata, the automata have the same reachable part and are therefore language-equivalent.

Let $M = (Q, \Sigma, \delta, q_0, F)$. Define $D := (2^Q, \Sigma, \delta', \{q_0\}, \{X \subseteq Q \mid X \cap F \neq \varnothing\})$, where δ' is defined as follows: $\delta(x, \sigma) = \{q' \mid q \in x \text{ and } (q, \sigma, q') \in \delta\}$.

The following algorithm computes the reachable part of D:

```
// Σ is given
function DETERMINIZE((Q, δ, q_0, F))
    W ← new Queue
    Add {q_0} to W
    Q′ ← ∅
    while W is not empty do
        x ← first(W)
        Remove x from W
        if x ∈ Q′ then
            continue
        end if
        Q′ ← Q′ ∪ {x}
        for σ ∈ Σ ∪ {ϵ} do
            x′ ← {q′ | ∃q, q ∈ x ∧ (q, σ, q′) ∈ δ}
            δ′(x, σ) ← x′
            Add x′ to Q
        end for
    end while
    return (Q′, δ′, {q_0}, {x ∈ Q′ | ∃q, q ∈ F ∩ x}).
end function
```

For an automaton with $|Q|$ states, this algorithm can produce a DFA with up to $2^{|Q|}$ states, and it may therefore have exponential running time. As it turns out, there are examples where this is the optimal solution – no smaller language-equivalent DFA exists (compare Problem 1.3).

2.2 Regular expressions

Let Σ be an alphabet. Another way to represent languages, i.e., specific subsets of Σ^*, are *regular expressions*. A regular expression is given by the following grammar:

$$R ::= \varnothing \mid \epsilon \mid \sigma \mid R \cdot R \mid R \cup R \mid R^*, \text{ where } \sigma \in \Sigma$$

Each regular expression can be interpreted as a set of words over Σ, as follows:

$$\begin{aligned}
L(\varnothing) &= \varnothing \\
L(\epsilon) &= \{\epsilon\} \\
L(\sigma) &= \{\sigma\} \text{ for all } \sigma \in \Sigma \\
L(R_1 \cdot R_2) &= \{w_1 \cdot w_2 \mid w_1 \in L(R_1) \text{ and } w_2 \in L(R_2)\} \\
L(R_1 \cup R_2) &= L(R_1) \cup L(R_2) \\
L(R^*) &= \{w_1 \cdot w_2 \cdots w_n \mid w_1, \ldots, w_n \in L(R)\}
\end{aligned}$$

Returning to the example from Figure 7, the language described is the language of the regular expression

$$R := n \cdot n \cdot n \cdot n \cdot n \cdot n \cdot n \cdot n \cdot (b \cup (h \cdot h \cdot g) \cup (n \cdot n \cdot (k \cup (n \cdot n \cdot n^*) \cup n \cdot h \cdot j)$$

Theorem 3. *Let $L \subseteq \Sigma^*$ be a language. Then the following are equivalent:*

1. *There is an FSM M such that $L = L(M)$.*
2. *There is a DFA D such that $L = L(D)$.*
3. *There is a regular expression R such that $L = L(R)$.*

$1 \Rightarrow 2$ is the subset construction from above. $3 \Rightarrow 1$ is Problem 1.2, and $2 \Rightarrow 3$ is Lemma 1.32 in [2].

2.3 FSMs and set operations

Let FSMs M_1 and M_2 be given. It turns out that there are FSMs $M_\cup$ and $M_\cap$ such that $L(M_\cup) = L(M_1) \cup L(M_2)$ and $L(M_\cap) = L(M_1) \cap L(M_2)$. Furthermore, for an FSM M, there is an FSM $\overline{M}$ such that $L(\overline{M}) = \overline{L(M)}$. The constructions are illustrated with a simple example in Figure 8.

Suppose $M_1 = (Q_1, \Sigma_1, q_1^0, F_1)$ and $M_2 = (Q_2, \Sigma_2, q_2^0, F_2)$. The construction of $M_\cup$ is straightforward: Let q_0 a fresh state, i.e., $q_0 \notin Q_1 \cup Q_2$. Then define

$$M_\cup := (Q_1 \cup Q_2 \cup \{q_0\}, \Sigma_1 \cup \Sigma_2, \delta_1 \cup \delta_2 \cup \{(q_0, \epsilon, q_0^1), (q_0, \epsilon, q_0^2)\}, F_1 \cup F_2).$$

It is easy to check that $L(M_\cup) = L(M_1) \cup L(M_2)$. The number of states of $M_\cup$ is $|Q_1| + |Q_2| + 1$.

To construct $\overline{M}$, we assume that $M = (Q, \Sigma, \delta, q_0, F)$ is a DFA. By the definition of DFAs, δ can be interpreted as a function $Q \times \Sigma \to Q$. In particular, this means that for every word w, there is a unique state $q \in Q$ such that $q_0 \xrightarrow{w} q$. For a given word $w \in \Sigma^*$, denote this unique state by q_w. Then $w \in L(M)$ if and only if $q_w \in F$. Define $\overline{M} := (Q, \Sigma, \delta, q_0, Q \setminus F)$. It is again a DFA, and it is easy to check that $\overline{M}$ accepts a word $w \in \Sigma^*$ iff $q_w \in Q \setminus F$. Thus, it accepts iff $q_w \notin F$, and therefore, if and only if M does not accept the word. Hence, $L(\overline{M}) = \overline{L(M)}$.

The construction of $M_\cap$ involves building the *product automaton*. Formally, the product automaton $M_1 \times M_2$ is constructed like this: Suppose $M_1 = (Q_1, \Sigma, q_1^0, F_1)$ and $M_2 = (Q_2, \Sigma, q_2^0, F_2)$. then $M_1 \times M_2 = (Q_1 \times Q_2, \Sigma, \delta, (q_1^0, q_2^0), F_1 \times F_2)$ where $((q_1, q_2), \sigma, (q_1', q_2')) \in \delta$ if one of the following conditions holds:

1. $(q_1, \sigma, q_1') \in \delta_1$ and $(q_2, \sigma, q_2') \in \delta_2$,
2. (q_1, ϵ, q_1'), $\sigma = \epsilon$ and $q_2 = q_2'$,
3. (q_2, ϵ, q_2'), $\sigma = \epsilon$ and $q_1 = q_1'$,

It is easy to check that $(q_1, q_2) \xrightarrow{w} (q_1', q_2')$ if and only if $q_1 \xrightarrow{w} q_1'$ and $q_2 \xrightarrow{w} q_2'$. Thus, $M_1 \times M_2$ accepts w if and only if both M_1 and M_2 accept w.

Algorithmically, the reachable part of $M_1 \times M_2$ can be constructed as follows:

// Σ is given
function PRODUCT($((Q_1, \delta_1, q_1^0, F_1), (Q_2, \delta_2, q_2^0, F_2))$)

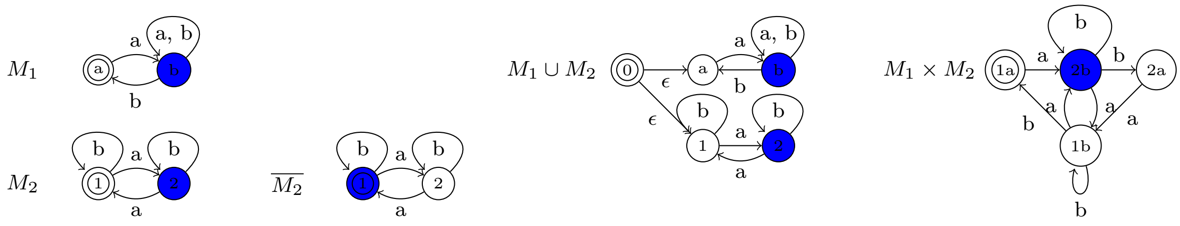


Figure 8: Examples of automaton constructions.

$W \leftarrow$ **new** `Queue`
Add (q_1^0, q_2^0) to W
$Q' \leftarrow \varnothing$
while W is not empty **do**
  $(q_1, q_2) \leftarrow \texttt{first}(W)$
  Remove (q_1, q_2) from W
  if $(q_1, q_2) \in Q'$ **then**
    continue
  end if
  $Q' \leftarrow Q' \cup \{(q_1, q_2)\}$
  for $\sigma \in \Sigma$ **do**
    for $q_1' : (q_1, \sigma, q_1') \in \delta_1$ **do**
      for $q_2' : (q_2, \sigma, q_2') \in \delta_2$ **do**
        $\delta' \leftarrow \delta' \cup \{((q_1, q_2), \sigma, (q_1', q_2'))\}$
        Add (q_1', q_2') to Q
      end for
    end for
  end for
  for $q_1' : (q_1, \epsilon, q_1') \in \delta_1$ **do**
    $\delta' \leftarrow \delta' \cup \{((q_1, q_2), \epsilon, (q_1', q_2))\}$
    Add (q_1', q_2) to Q.
  end for
  for $q_2' : (q_2, \epsilon, q_2') \in \delta_2$ **do**
    $\delta' \leftarrow \delta' \cup \{((q_1, q_2), \epsilon, (q_1, q_2'))\}$
    Add (q_1, q_2') to Q.
  end for
end while
return $(Q', \delta', (q_1^0, q_2^0), F_1 \times F_2)$.
end function

The product automaton has at most $|Q_1| \cdots |Q_2|$ states, and the algorithm can be shown to have running time $O(|Q_1| \cdot |Q_2| \cdot |\Sigma|)$.

References

[1] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[2] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996.