Supplementary file

Dynamical modeling and analysis of large cellular regulatory networks

D. Bérenguier, C. Chaouiya, P.T. Monteiro, A. Naldi, E. Remy, D. Thieffry, L. Tichit

1 Description

The Hierarchical Compaction Algorithm (HCA) builds a Hierarchical Transistion Graph (HTG) using the logical regulatory graph as an input, as defined in section III. It is implemented in GINsim (`http://ginsim.org/beta`) as a new functionality.

It is based on Tarjan's algorithm[1] which is a reference algorithm for finding the Strongly Connected Components (SCC) of a given graph. Following this approach, we propose an algorithm that compacts a State Transition Graph (STG) on the fly with a complexity of the same order as Tarjan's.

Starting from an initial state, Procedure 1 (**DFS**) relies on a Depth-First Search and explores a path, eventually reaching either a stable state or a backward arc leading to an already visited state, thus emphasizing a cycle that may belong to a larger strongly connected component. When a cycle has been discovered, Procedure 2 (**HCA**) is called. It relies on Procedure 3 (**compactSCC**) to replace the states of the cycle by a single hierarchical node. It also updates the σ-image of this HTG node. If it is a trivial transient component and if another component built so far shares the same σ-image, Procedure 4 (**compactIC**) merges them together, ensuring that there are no two components with the same σ-image. When HTG nodes are created, the role of Procedure 5 (**updateArcs**) is to add arcs to the graph. In case we start from a set of initial states, we can trivially extend this algorithm using Procedure 6 (**Init**).

[1] R. Tarjan, *SIAM J. Comput.* 1, 146-60 (1972)

2 Notations and conventions

- Variable $currentIndex$, stack P, tables $index$ and $lowLink$ are used to find the strongly connected components. Please refer to the description of Tarjan's original algorithm for more information concerning their semantics. They are initialized to 0 or $\emptyset$.
- s and s_i denote **states** of the state transition graph, whereas X and Y denote (hierarchical) **nodes** of the HTG (*i.e.* set of states).
- Function $successors(s : state)$ returns the set of states s_i such that $T(s, s_i) = 1$.
- Map *HTGarcs* associates each node to a set of adjacent HTG nodes.
- Map *HTGnodes* associates each state to the HTG node it belongs to.

3 Complexity

Please refer to main manuscript for notations. In addition to the $|S| + |T|$ operations of Tarjan's DFS, we add, for each state, the computation of its σ-image (constant time) and either one of two procedure calls (**compactSCC** or **compactIC**).

- Procedure **compactSCC** is called whenever a set of states belonging to a strongly connected component has been found. This procedure calls **updateArcs** once for every state in the strongly connected component.

- Procedure **compactIC** is called once on each trivial transient node (*i.e.* containing a single state s). This procedure calls **updateArcs** once. All other operations, in particular access to the irreversible nodes, are performed in constant time. Thus, the complexity of **compactIC** is the same as **updateArcs**.

- Procedure **updateArcs** is called exactly once for every state, either through **compactIC** (if it is a trivial transient state) or through **compactSCC** (if the state belongs to a strongly connected component), *i.e.* in total, once for every state ($= |S|$ times). This procedure performs $|HTGarcs(X)|$ operations for each node $X = \{s\}$. In the worst-case (a state transition graph reduced to a single SCC containing an Hamiltonian path), $|HTGarcs(X)| = d_{STG}(s)$, where $d_{STG}(s)$ is the degree of state s in the state transition graph. Hence, the complexity of procedure **updateArcs** is $O(\bar{d}_{STG})$ where $\bar{d}_{STG}$ is the average degree of the state transition graph.

Given that $O(|S|\times\bar{d}_{STG}) = O(|T|)$, the complexity of the hierarchical clustering algorithm is $O(|S|+|T|)$.

4 Optimizations and implementation

We have added two optimizations to the implementation, resulting in better performance, time- as well as space-wise.

- Hierarchical components encompass a set of states. Ordered Multivalued Decision Diagrams (OMDD) are well suited to save space in storing such sets of states[2]. The main drawback is that, to check if a state has been visited and to recover the HTG node it belongs to, we have to look through each component. Hence, the complexity increases to $O((|\mathscr{C}| + |\mathcal{I}|) \times |G|)$.

- When a backward arc is found during the DFS, it means that the state on the top of the stack and the one pointed by the backward arc belong to the same cycle, possibly a part of a larger SCC. As we are interested in keeping the stack as small as possible, we compress the cycles as soon as they are discovered during the DFS. To that end, the stack contains elements representing either a state or a (partial) strongly connected component. This decreases tremendously the average size of the stack. This improvement is presented in Procedure 7 (**DFS2**). It replaces Procedure 1 and Procedure 2.

[2]Willadsen and Wiles, J. Theor. Biol **249**, 749 (2007)

Procedure 1 DFS. Mostly a clone af Tarjan algorithm, looks for SCCs. If a SCC is found, Procedure **HCA** is called.

```
Procedure DFS (s: state)
index[s] ← currentIndex
currentIndex ← currentIndex + 1
P.push(s)
for all state s_i ∈ successors(s) do
  if s_i has not been visited then
    HTGarcs[{s}].add(s_i)
    HTGarcs[{s_i}] ← ∅
    lowLink[s_i] ← currentIndex
    call DFS(s_i)
    lowLink[s] ← min(lowLink[s], lowLink[s_i])
  else if s_i ∈ P then
    lowLink[s] ← min(lowLink[s], index[s_i])
  end if
end for
if lowLink[s] = index[s] then
  X ← ∅
  repeat
    t ← P.pop()
    X.add(t)
  until t = s
  call HCA(X)
end if
```

Procedure 2 HCA. If node X is a non-trivial SCC, Procedure **compactSCC** is called. Its σ-image is computed. If this X is a trivial transient node, and if another node built so far shares the same σ-image, Procedure **compactIC** is called.

```
Procedure HCA (X: node)
if |X| > 1 then
   call compactSCC(X)
   σ[X] ← X
else if X is an attractor {s} then
   σ[X] ← X
   HTGnodes[s] ← X
else
   σ[X] ← ∅
end if
for all Y ∈ HTGarcs[X] do
   σ[X] ← σ[X] ∪ σ[Y]
end for
if X is a trivial transient node then
   call compactIC(X)
end if
```

Procedure 3 compactSCC. Replaces the states of a SCC by a single node.

```
Procedure compactSCC (X: complex SCC node)
HTGarcs[X] ← ∅
for all state s ∈ X do
   HTGnodes[s] ← X
   call updateArcs({s}, X)
end for
HTGarcs[X] ← HTGarcs[X] \ {X}
```

Procedure 4 compactIC. For a given trivial transient node X, **compactIC** merges X with the node Y such that $\sigma(X) = \sigma(Y)$, if such a node exists.

```
Procedure compactIC (X: trivial transient node {s})

if ∃ node Y ≠ X s.t. σ[Y] = σ[X] then
   Y ← Y ∪ X
   call updateArcs(X, Y)
else
   HTGnodes[s] ← X
   call updateArcs(X, X)
end if
```

Procedure 5 updateArcs. Adds the outgoing arcs of trivial node X (i.e. state) to node HN which encompasses it.

```
Procedure updateArcs (X: trivial node, HN: node)

for all node Y ∈ HTGarcs[X] do
   if |Y| > 1 then
      HTGarcs[HN].add(Y)
   else /* Y is a trivial node */
      HTGarcs[HN] ← HTGarcs[HN] ∪ {HTGnodes[Y]}
   end if
end for
```

Procedure 6 Init. Initialisation and Depth First Search algorithm in the case of several initial states

```
stack P ← ∅
currentIndex ← 0
for all s ∈ {initial states} do
   if s has not been visited then
      HTGarcs[{s}] ← ∅
      lowLink[s] ← currentIndex
      call DFS(s)
   end if
end for
```

Procedure 7 DFS2. Modification of procedure DFS in order to compress SCCs as soon as possible

```
Procedure DFS2 (s: state)
index[s] ← currentIndex
currentIndex ← currentIndex + 1
P.push(s)
for all state s_i ∈ successors(v) do
   if s_i has not been visited then
      HTGarcs[{s}].add(s_i)
      HTGarcs[{s_i}] ← ∅
      lowLink[s_i] ← currentIndex
      call DFS(s_i)
      lowLink[s] ← min(lowLink[s], lowLink[s_i])
   else if s_i ∈ P or SCC[s_i] ∈ P then
      X ← ∅
      repeat
         t ← P.pop()
         X.add(t)
      until t = s
      call compactSCC(X)
      σ[X] ← X
      P.push(X)
      lowLink[X] ← min(lowLink[s], index[s_i])
   end if
end for
if lowLink[s] = index[s] then
   X ← P.pop()
   if X is an attractor then
      σ[X] ← X
      HTGnodes[s] ← X
   else
      σ[X] ← ∅
   end if
   for all Y ∈ HTGarcs[X] do
      σ[X] ← σ[X] ∪ σ[Y]
   end for
   if X is a trivial transient node then
      call compactIC(X)
   end if
end if
```